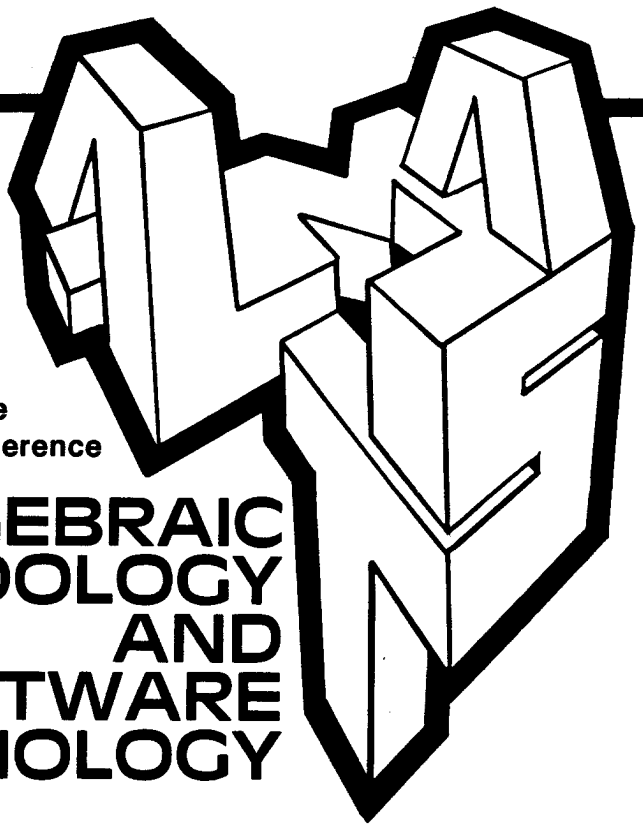


Proceedings of the
First International Conference

ALGEBRAIC METHODOLOGY AND SOFTWARE TECHNOLOGY

May 22-24, 1989

The University of Iowa
Iowa City, Iowa

Mathematics well-applied illuminates rather than confuses

First International Conference on Algebraic Methodology and Software Technology, AMAST[1]

May 22-24, 1989, Iowa City, Iowa, USA

Organizing Committee:

Conference Chairs	A. Fleck	University of Iowa Department of Computer Science Iowa City, IA 52242, USA
	W.A. Kirk	University of Iowa Department of Mathematics Iowa City, IA 52242, USA
Program Committee Chair	T. Rus	University of Iowa Department of Computer Science Iowa City, IA 52242 USA
Committee Members	W.S. Hatcher	Université Laval, Canada
	G. Hotz	Universität des Saarlandes, West Germany
	D. Ionescu	University of Ottawa, Canada
	E. Madison	University of Iowa, USA
	M. Main	University of Colorado, USA
	M. Mislove	Tulane University, USA
	M. Muralidharan	University of Iowa, USA
	G. Nelson	University of Iowa, USA
	C. Rattray	University of Stirling, Scotland
	D. Schmidt	Kansas State University, USA

[1]This conference was sponsored by grants from the Office of Naval Research, XEROX Corporation – Webster Research Center, Rochester, N.Y., Departments of Computer Science and Mathematics of the University of Iowa, Iowa City

Organizing Committee's Message

Past research on mathematical foundation of computer science has focused mostly on the study of the mathematics of the software objects and very little has been done to develop software objects on a mathematical basis. Languages, programs, and the process of program execution have been identified as fundamental objects of study of computer science as a discipline. Moreover, algebraists and computer scientists have begun to relate the abstractions in computer science to the process of abstract representation in universal algebra. A strong trend of applying universal algebras as the mathematical foundation of computer science is in vogue. In pursuing this trend we need to observe however that there are differences between the objects and methods used in universal algebra and computer science. While abstractions used in universal algebra represent behavior of ideal (mathematical) objects, abstractions in computer science represent behavior of real (physical) objects. While the ideal character of the abstractions in universal algebra allows systematic approaches of their specifications and development of formal notations naturally suited to handling them, computer science develops formal notations to denote real objects that are rarely formally specified. While an algebraic language accommodates naturally *semantics*, *syntax*, and *semantics*$\leftrightarrow$*syntax* association of the algebraic abstractions, the *syntax* and the *semantics* of a computer language are specified by different mechanisms and their association into a language is artificial.

The similarities of the abstractions handled in universal algebra and computer science lead to the development of new mathematical theories. Our conjecture is that keeping in view also the differences between the abstractions used in universal algebra and computer science, new mathematics can be created that will facilitate the construction of the (software) objects arising in computer science. The goal of this conference is to consolidate this conjecture, looking at algebraic methodology as a foundation for software technology and showing that universal algebra provides a practical mathematical alternative to ad hoc approaches used in software development. This idea was well received by the international and national industrial and research communities reflecting the desire for the development of software technology on a mathematical basis. Therefore, unlike other conferences on mathematical foundations of computer science, in which usually the mathematics is enriched with new theories originated in computer science, the submissions to this conference indeed show developments in computer science that originate in mathematics.

From the 89 submissions we had we could deduce a large spectrum of use of algebraic methods as mathematical basis of the new software technology. The major directions extracted from these submissions formed the guidelines for the organization of the technical program of the conference and are:

- Formalizing the concept of a process and modeling the computer activity as an algebra of processes.
- New alternatives for computer language specification and implementation based on universal algebras rather than grammars.
- Formal models of the parallel and distributed systems.
- Type algebra allowing the extension of the class of first order values.
- General application of the algebra to software development and maintenance.

There are 5 invited talks from the following distinguished speakers: R. Constable (Cornell University, USA), W. F. Lawvere (State University of New York at Buffalo, USA), J. Meseguer (SRI International, USA), M. Nivat (The University of Paris VII, France), and E. Wagner (IBM Thomas J. Watson Research Center, USA). Our special thanks to them, for taking time from their busy schedules and accepting our invitation to give a talk. They will provide special insight into current areas of research reflecting the theme of the conference.

There are 27 contributed papers in the proceedings. Of the 89 papers submitted in response to the call for papers, 27 were selected for presentation at the conference. The selection process was carried out by the program committee along with the following additional reviewers: Maria Zamfir-Bleyderg (Kansas State University), Steve Bruell (University of Iowa) and Mahesh Dodani (University of Iowa). Each paper was reviewed by at least two reviewers and the final selection was based on the composite scores, originality and relevance to the theme of the conference. We wish to thank all the reviewers for their time and effort. In addition, we wish to thank all those who submitted papers.

The conference would not have been possible without the generous support of the following sponsors: Office of Naval Research, XEROX Corporation - Webster Research Center, and Departments of Computer Science and Mathematics of the University of Iowa. We would like to thank all of them for their financial assistance and interest. Finally, we would like to thank our secretaries Cyndi, Beth, Julie and Margaret, for their assistance in secretarial matters.

An issue of the international journal "Theoretical Computer Science" will be dedicated to this conference. All participants at the conference are invited to submit papers to this issue. The submissions should be sent, no later than August 1, 1989, to:

AMAST Organizing Committee
Department of Computer Science
The University of Iowa
Iowa City, IA 52242, USA

Conference Program

Sunday, May 21, 6:30-8:30pm **Reception**

Monday, May 22, 8:00-8:45 **Registration**

Monday 22, 8:45-9:00 **Welcome, Introduction:** Fleck, A., Conference Chair.

Monday 22, 9:00-12:30: Session 1 **Process Algebra**
(Chair: Nelson, G., The University of Iowa, Iowa, USA)

- 9:00-10:00 Invited talk: *Minimal Finite Transition Systems*, Nivat, M., Université Paris VII, France.
- 10:00-10:30 Baeten, J., *Algebra of Communicating Processes.*

10:30-11:00 Coffee break

- 11:00-11:30 Crew, R. F., *Parameterized Process Category.*
- 11:30-12:00 Pigozzi, D., *Equality-Test and If-Then-Else Algebras: Axiomatization and Specification.*
- 12:00-12:30 Benson, D. B., Iyer, R.R., *Algebraic Structure of Petri-Nets and Nondeterminism.*

12:30-2:00 Lunch

Monday 22, 2:00-6:00 Session 2 **Algebraic Methods for Language Specification**
(Chair: Hatcher, W.S., Université Laval, Quebec, Canada)

- 2:00-3:00 Invited talk: *Display of Graphics and their Applications, as Exemplified by 2-Categories and Hegelian "Taco"*, William F. Lawvere, Buffalo University, USA.
- 3:00-3:30: Bidoit, M., *The Stratified Loose Semantics: An Attempt to Provide an Adequate Algebraic Model of Modularity.*
- 3:30-4:00 Jacobs, D., Ehrig, H., Fey, W., Hansen H., Lowe M., *Algebraic Concepts for the Evolution of Module Families*

4:00-4:30 Coffee break

- 4:30-5:00 Bradley L., *An Algebraic Approach to the Early Stages of Language Design.*
- 5:00-5:30 Talcott, C. L., *Algebraic Methods in Programming Language Theory.*
- 5:30-6:00 Parpucea, I., *Dynamic Extension of Programming Language Semantics.*
- 7:30 Social hour

Tuesday 23, 8:30-12 Session 3 **Parallel and Distributed Processing**
(Chair: Cornell, A., Brigham Young University, Utah, USA)

- 8:30-9:30 Invited talk: *General Logics*, Jose Meseguer, SRI, USA.
- 9:30-10:00 Logrippo, L., *LOTOS: An Algebraic Specification Language for Distributed Systems.*

10:00-10:30 Coffee break

- 10:30-11:00 Miller, S., Kuhl, J., *Modeling Distributed Systems as Distributed Data Types.*
- 11:00-11:30 Ionescu, D., Wen, L., *A Formal Mathematical Model for Detecting the Subroutine Dependencies: A Logic Programming Approach.*
- 11:30-12 Martin, G.A.R., Norris, M.T., Everett, R.P., Shields, M.W., *The CCS Interface Equation – An Example of Specification Construction Using Rigorous Techniques.*

12-1:30 Lunch

Tuesday 23, 1:30-5:30 Session 4 **Types, Polymorphism and λ-Calculus**
(Chair: Main, M., University of Colorado at Boulder, Colorado, USA)

- 1:30-2:30 Invited talk: *Implementing Mathematics as an Approach for Formal Reasoning*, R. Constable, Cornell University, USA.
- 2:30-3:00 Hatcher, W.S., Tonga, M., *Pairings on Lambda Algebras.*
- 3:00-3:30 Zhang, H., *Constructor Models as Abstract Data Types.*

3:30-4:00 Coffee break

- 4:00-4:30 Riecke, J.G., Bloom, B., *LCF Should be Lifted.*
- 4:30-5:00 Scollo, G., Manca, V., Salibra, A., *DELTA: A Deduction System Integrating Equational Logic and Type Assignment.*
- 5:00-5:30 Janicki, R., Müldner, T., *On Algebraic Transformations of Sequential Specifications.*
- 6:30 Banquet (Buses depart)

Wednesday 24, 8:30-12 Session 5 **Algebraic Software Development**
(Chair: Schmidt, D., Kansas State University, Kansas, USA)

- 8:30-9:30 Invited talk: *An Algebraically Specified Language for Data Directed Design,* Eric Wagner, IBM Thomas J. Watson Center, USA.
- 9:30-10:00 Rattray, C.I.M., *Modeling the Software Process.*

10:00-10:30 Coffee break

- 10:30-11:00 Wells, C., *Path Grammars.*
- 11:00-11:30 Pratt, V., *Enriched Categories and Floyd-Warshall Connection.*
- 11:30-12 Dauchet, M., Tison, S., *Finite Automata, Algorithms and Software Design.*

12-1:30 Lunch

Wednesday 24, 1:30-5:00 Session 6 **Algebraic Semantics of Programs**
(Chair: Rattray, C.I.M., University of Stirling, Stirling, Scotland)

- 1:30-2:00 Schmidt, D., Even, S., *Category-Sorted Algebra-Based Action Semantics.*
- 2:00-2:30 Vidal, D., *The De Bruijn Algebra*
- 2:30-3:00 Oguztuzun, H.M., *A Game Characterization of the Observational Equivalence of Processes.*

3:00-3:30 Coffee break

- 3:30-4:00 Wijland, W.P., Van Glabbek R.J., *Refinement in Branching Time Semantics.*
- 4:00-4:30 Kent, R.E., *Dialectical Program Semantics.*
- 4:30-5:00 Concluding remarks

ON MINIMAL FINITE TRANSITION SYSTEMS[1]

Maurice Nivat, L.I.T.P., University Paris VII

Introduction

The theory of finite automata, which we shall rather call finite transition systems is certainly the oldest chapter of theoretical computer science. Most of the algebraic results come from the consideration of deterministic automata since there is in each class of automata recognizing the same language a "minimal" one which has indeed the smallest number of states but also is the image in a morphism of all the automata in the class. Every knows how to compute this minimal automaton which is unique from either a given automaton or a rational expression representing its language. Moreover this automaton is closely linked with the syntactic monoid ant the so called Nerode equivalence which is the smallest right-regular equivalence relation which saturates the language.

The situation is entirely different if one considers non deterministic finite transition systems : it is immediate that one may have two equivalent finite transition systems, recognizing the same language, which have the same smallest number of states and which cannot be mapped by morphisms onto a same smaller finite transition system recognizing the same language. The study of finite transition systems has been greatly exhauced in recent years by the construction of various models of parallel or distributed systems. Among these models the calculus CCS of Robin Milner has been extremely

---

[1] Part of an unfinished paper

influential. In this paper we borrow many ideas from R. Milner's work especially the notion of observational equivalence, and its reformulation by André Arnold and Anne Dicky using morphisms which we call MR-morphisms in the sequel. It is a restricted notion of equivalence based on the idea that two systems are equivalent if each one simulates the other one. The simulation of S_1 by S_2 means that for every possible behaviour of S_1 the exists a simulating behaviour of S_2 with the property that at each instant of time S_2 will be in a state in which it can perform all the actions which S_1 may perform and only those.

We introduce a more general notion of equivalence using the family of functional morphisms : a functional morphism of S_1 onto S_2 has the property that every behaviour (or computation) of S_2 is the image of a behaviour of S_1 . We thus characterize the equivalence $S_1 \sim_F S_2 \Leftrightarrow \mathcal{L}(S_1) = \mathcal{L}(S_2)$ where $\mathcal{L}(S)$ is the global language of a system ie the set of traces of all the computations from any state to any state. The characterization is extremely similar to the characterization of MR morphisms.

In a last chapter we consider deterministic systems and surprisingly we discover that under various assumptions (right separatedness, strong connectivity) the functional morphisms happen to be MR morphisms.

I. Morphisms and quotients of Finite Transition Systems

We shall use the following definitions and notations.

If $S = \langle Q,T \rangle$ is a FTS and R is an equivalence relation on Q we define the quotient of S by R as the FTS $S/R = \langle Q/R, T/R \rangle$ where

Q/R is the set of equivalence classes modulo R.
(We denote $[q]_R$ or simply $[q]$ the equivalence class of q).

T/R is the set of transitions

$$T/R = \{([q] \xrightarrow{a} [q']) \mid \exists\, q_1 \in [q]\ \exists\, q'_1 \in [q'] : (q_1 \xrightarrow{a} q'_1) \in T.$$

A quotient of S is also the image of S by a morphism : a morphism h of S is defined by a mapping h of Q onto a finite set $h(Q)$, the image $h(S)$ being the FTS

$h(S) = \langle h(Q), h(T) \rangle$ where
$h(T) = \{(h(q) \xrightarrow{a} h(q')) \mid (q \xrightarrow{a} q') \in T\}$

Clearly if R is an equivalence relation, the canonical mapping h_R of S onto Q/R is a morphism of S onto S/R and we can write $h_R(S) = S/R$.

Conversely a mapping h of Q onto $h(Q)$ defines the canonical equivalence relation R_h such that

$$q \sim_{R_h} q' \Leftrightarrow h(q) = h(q')$$

and the morphic image h(S) is isomorphic to the quotient S/R_h.

A computation of the transition system $S = \langle Q,T \rangle$ is denoted $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \quad \xrightarrow{a_n} q_n$ assuring that for all $i \in [n]$ $q_{i-1} \xrightarrow{a_i} q_i$ is a transition of T.

We denote Comp(S, q, q') the set of all computations

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \quad \xrightarrow{a_n} q_n \text{ such that } q_0 = q \text{ and } q' = q_n.$$

The sets Comp(S, q, Q) and Comp(S,Q,q') are defined as

$$\text{Comp}(S, q, Q) = \cup \{\text{Comp}(S, q, q') \mid q' \in Q\}$$
$$\text{Comp}(S, Q, q') = \cup \{\text{Comp}(S, q, q') \mid q \in Q\}$$

The trace of the computation $c = (q_0 \xrightarrow{a_1} q_1 \rightarrow .. \xrightarrow{a_n} q_n)$ is $\lambda(c) = a_1a_2 ...a_n$ which is a word in A^+.

And we define the following sets of traces

$$L(S, q, q') = \lambda(\text{Comp}(S, q, q')) \text{ if } q \neq q'$$
$$= \{\varepsilon\} \cup \lambda(\text{Comp}(S, q, q')) \text{ if } q = q'$$

(we assume that there is always an empty computation $q \xrightarrow{\varepsilon} q$ from any state q to itself)

$$L(S, q, Q) = \cup \{L(S, q, q') \mid q' \in Q\}$$
$$L(S, Q, q) = \cup \{L(S, q, q') \mid q \in Q\}$$

And eventually we denote $\mathcal{L}(S)$ the global language of S

defined by $\mathcal{L}(S) = \cup \{L(S, q, q') \mid q, q' \in Q\}$.

We note that $\mathcal{L}(S)$ is factorial ie satisfies

- for all $f_1, f_2, f_3 \in A^+$ $f_1f_2f_3 \in \mathcal{L}(S) \Rightarrow f_2 \in \mathcal{L}(S)$.

The following properties are immediate

Let h be a morphism of S onto h(S) and $c = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_1} \xrightarrow{a_n} q_n$ be a computation of S then $h(c) = h(a_0) \xrightarrow{a_1} h(a_1) \rightarrow \ldots \xrightarrow{a_n} h(q_n)$ is a computation of h(S).

Thus we have

Property *For every morphism h of a transition system S = <Q,T> one has h (Comp (S, q, q')) $\subseteq$ Comp (h (S), h (q), h (q'))*

$L(S, q, q') \subseteq L(h(S), h(q), h(q'))$ and

It is not generally true that these inclusions are equalities and the morphismes for which they are indeed equalities will play a major role in the sequel.

Definition *The morphism h of S onto h (S) is functional iff every computation in h (S) is the image under h of a computation in S.*

Equivalence of FTS

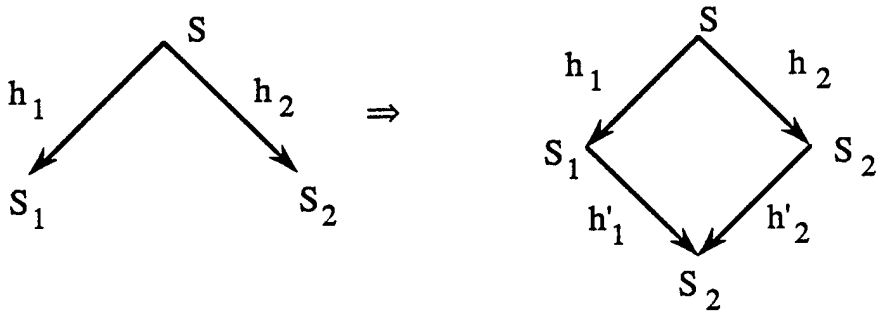
Several notions of equivalences will be considered in the present paper.

Three of them will be attached to families of morphisms which have the Church-Rosser property.

The family H of morphisms is a Church-Rosser family (or is CR) iff

- the identity belongs to H
- H is closed under composition
- if S is a FTS and h_1, h_2 are the H-morphism of S onto $h_1(S) = S_1$ and $h_2(S) = S_2$ them there exist two H-morphisms h'_1 and h'_2 such that $h'_1(S_1) = h'_2(S_2)$.

This last property can be visualized by the diamond diagram

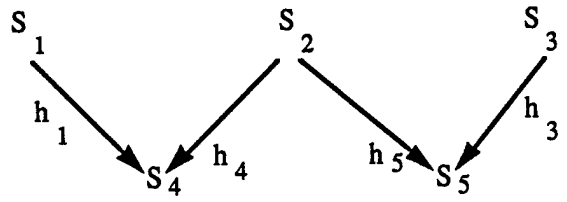


Property I.2 *If H is a Church-Rosser family of morphisms then the relation* $\equiv_H$ *defined by*

$$S_1 \equiv_H S_2 \Leftrightarrow \exists\, h_1, h_2 \in H \qquad h_1(S_1) = h_2(S_2)$$

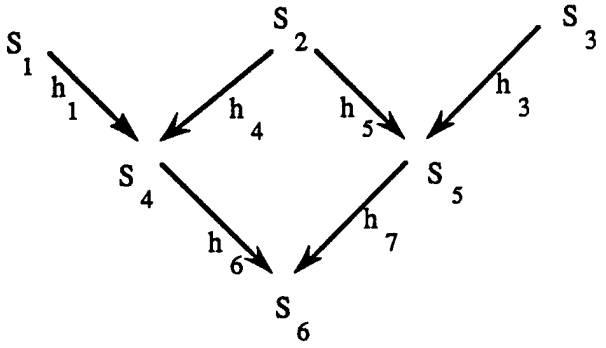
is an equivalence relation between transition systems which we call the lower H-equivalence.

Proof It is an immediate consequence of the CR property. If $S_1 \equiv_H S_2$ and $S_2 \equiv_H S_3$ we have the following diagram



which we can complete

We get



and this H is closed by composition S_6 is the image of S_1 and S_3 by the two H-morphisms $h_1 \circ h_6$ and $h_3 \circ h_4$ (we say also that S_6 is a common H-quotient of S_1 and S_3).

Property I.3 *The family Mor of all morphisms is CR*

Proof We consider the two morphisms h_1 and h_2 of S onto $h_1(S) = S_1$ and $h_2(S) = S_2$.

We denote R_1 and R_2 the two equivalence relations on Q corresponding to h_1 and h_2 and we call R the smallest equivalence relation on Q containing R_1 and R_2.

We recall that R is the transitive closure of $R_1 \cup R_2$ which means that

$q\ R\ q' \Leftrightarrow \exists\ q_0,\ldots,q_h$ such that

$q_0 = q$, $q' = q_h$ and for all $i \in [h]$ $q_{i-1}\ R_1\ q_i$ or $q_{i-1}\ R_2\ q_i$.

The morphism h corresponding to R maps S onto S' and it is clear that h can be factorized in $h_1 \circ h'_1$ and $h_2 \circ h'_2$ since the equivalence classes modulo R are union of equivalence classes modulo R_1 (resp. R_2). □

Unfortunately the lower equivalence $\equiv_{Mor}$ is not very interesting, since we have the following

Property I.4 *Let A' be a subset of A and denote $S_{A'}$ the FTS with only one state q_0 and the set of transitions*

$$\{q_0 \xrightarrow{a} q_0 \mid a \in A'\}$$

Then the mapping h of $S = <Q,T>$ defined by $h(q) = q_0$ for all $q \in Q$ is a morphism of S onto $S_{A'}$ iff $A' = \lambda(T) = \{a \in A \mid \exists\ q, q' \in Q\ (q \xrightarrow{a} q') \in T\}$.

Proof obvious

And the property I.3 implies immediately that

$$S_1 \equiv_{Mor} S_2 \Leftrightarrow \lambda(T_1) = \lambda(T_2).$$

It is clear since one can map S_1 onto S_{A_1} if $A_1 = \lambda(T_1)$ and S_2 onto S_{A_2} if $A_2 = \lambda(T_2)$ and clearly $S_{A_1} \equiv_{Mor} S_{A_2}$ iff $A_1 = A_2$.

We now introduce a more interesting family of morphisms denoted MR : a morphism in MR is called a right Milner

morphism, since the equivalence associated with MR has been first considered by R. Milner and we shall define later a family of left Milner morphisms.

Definition The morphism h of S is a right Milner morphism (or an MR morphism) iff it satisfies

$\forall\, q, q', q_1 \in Q\,, \forall\, a \in A$

$(q \xrightarrow{a} q_1) \in T$ and $h(q') = h(q)$ imply $\exists\, q'_1 : (q' \xrightarrow{a} q'_1) \in T$ and $h(q'_1) = h(q_1)$

Property I.5 *The family MR of right Milner morphism is a Church Rosser family.*

Proof It is immediate from the proof of property I.2. We define R as the transitive closure of R_1 and R_2 and we assume that $q\, R\, q'$ ie there exist $q_0, ..., q_h$ such that $q_0 = q$, $q' = q_k$ and for all $i \in [h]$ $q_{i-1}\, (R_1 \cup R_2)\, q_i$. Then we assume the existence of a transition $q_0 \xrightarrow{a} \overline{q}_0$: the morphisms h_1 and h_2 being RM we know that there exists $\overline{q}_1$ such that $q_1 \xrightarrow{a} \overline{q}_1$ and $\overline{q}_1\ R_1\ \overline{q}_0$ or $\overline{q}_1\ R_2\ \overline{q}_0$ according to whether $q_0\, R_1\, q_1$ or $q_0\, R_2\, q_1$.

By an easy induction on k we can find $\overline{q}_h$ such that $q_k \xrightarrow{a} \overline{q}_k$ and $\overline{q}_k\ R\ \overline{q}_0$, thus proving that h_R is RM. It is easy then to prove there exist two RM morphisms h'_1 and h'_2 such that $h_R = h_1 \circ h'_1 = h_2 \circ h'_2$.□

We have to remark

Property I.6 *Every MR morphisms is a functional morphism.*

Proof This is an immediate consequence of the definition. Every computation of length 1, ie every computation of the form $h(q) \xrightarrow{a} h(q')$ in $h(S)$ is the image under h of some computation $q_1 \xrightarrow{a} q'_1$ in S by the very definition of a morphism. Assume we have proved that every computation of length n in S and consider a computation of length $n+1$ in $h(S)$

$$\bar{q}_0 \xrightarrow{a_1} \bar{q}_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} \bar{q}_n \xrightarrow{a_{n+1}} \bar{q}_{n+1}$$

By induction there exists a computation in S

$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n$ such that for all $i \in \{0,\ldots,n\}$ $h(q_i) = \bar{q}_i$. then since h is MR and we know that there exist a transition $\bar{\bar{q}} \xrightarrow{a_{n+1}} \bar{\bar{q}}_{n+1}$ with $h(\bar{\bar{q}}_n) = \bar{q}_n$ and $h(\bar{\bar{q}}_{n+1}) = \bar{q}_{n+1}$ from the fact that $h(q_n) = h(\bar{\bar{q}}_n)$ we can infer the existence a a transition $q_n \xrightarrow{a_{n+1}} q_{n+1}$ for some q_{n+1} satisfying $h(q_{n+1}) = \bar{q}_{n+1}$. Thus $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n \xrightarrow{a_{n+1}} q_{n+1}$ is a computation in S whose image by h is the computation $\bar{q}_0 \xrightarrow{a_1} \bar{q}_1 \xrightarrow{a_2} \ldots \xrightarrow{a_{n+1}} \bar{q}_{n+1}$. □

Partition corresponding to right Milner morphisms

For any given morphism h of S one has a partition of the set of states Q which is the partition in equivalence classes modulo h. And conversely if $Q = Q_1 \cup \ldots \cup Q_h$ is a partition of Q. It defines a morphism of S : one maps all the states in each component Q_i onto a single state $i \in [h]$.

In this paragraph we characterize the MR-partitions corresponding to MR morphism.

Restrictions of a transition system

If Q' is a subset of Q we define the restriction of $S = \langle Q,T \rangle$ to Q' as the system $S \mid Q'$ given by

- its set of states Q'
- the set of transitions $T \mid Q' = \{t \in T \mid \alpha(t) \in Q' \text{ and } \beta(t) \in Q'\}$

The subset Q' of Q is said to be monoïdal iff

$$\mathcal{L}(S \mid Q') = \lambda(T \mid Q')^*$$

The set $\lambda(T \mid Q')^*$ is the subset of the alphabet A formed by all the letters which label a transition of $T \mid Q'$ or equivalently all the letters which label a transition in T whose origin and extremity belong to Q'. We allow $\lambda(T \mid Q')$ to be empty : then $\mathcal{L}(S \mid Q) = \{\varepsilon\}$. The first remark is

Property I.7 *Each component of an MR partition of Q is monoïdal.*

Proof Let $Q = Q_1 \cup \ldots \cup Q_k$ be the MR partition corresponding to the MR-morphism h which maps for all $i \in [k]$ all the elements of Q_i onto the state i of $h(Q) = [h]$.

Clearly the set of transitions of $h(S)$ contains all the transitions $i \xrightarrow{a} i$ for all $i \in [h]$ and $a \in \lambda(T \mid Q_i)$.

In fact $h(T) \mid \{i\} = \{i \xrightarrow{a} i \mid a \in \lambda(T \mid Q_i)$.

This proves that $\mathcal{L}(S \mid Q_i) \subseteq \lambda(T \mid Q_i)^* = \mathcal{L}(h(S) \mid \{i\})$.

The reverse inclusion comes from the fact that the MR morphism h is also functional : every computation of $h(S) \mid \{i\}$ is the image of a computation of $S \mid Q_i$ whence

$$\mathcal{L}(h(S) \mid \{i\}) \subseteq \mathcal{L}(S \mid Q_i). \qquad \square$$

In fact we can prove a stronger property : in the same situation as above one has

$$L(S \mid Q_i , q_i , Q_i) = \lambda(T \mid Q_i)^*$$

For every computation c' in $h(S) \mid \{i\}$ and every state q_i in Q_i one can find a computation c in $S \mid Q_i$ such that $h(c) = c'$ and $\alpha(c) = q_i$, this is just a new application of the MR condition. $\square$

We can call strongly right monoïdal a subset of Q such that for all $q_i \in Q_i$ $L(S \mid Q_i, q_i, Q_i) = \lambda(T \mid Q_i)^*$ and then state the

Theorem I.1

The partition $Q_i = Q_1 \cup Q_2 \cup \ldots \cup Q_k$ is an MR partition iff

- for all $i \in [h]$ Q_i is strongly right monoïdal

- for all $i,j \in [h]$ if there exists a transition t such that $\alpha(t) \in Q_i$, $\beta(t) \in Q$, and $\lambda(t) = a$ then for all $q_i \in Q_i$ there exists a transition t' such that $\alpha(t') = q_i$, $\beta(t') \in Q_j$ and $\lambda(t) = a$.

Proof The only if part follows immediately from property I.7 and the MR condition.

Conversely the morphism corresponding to the partition is MR.

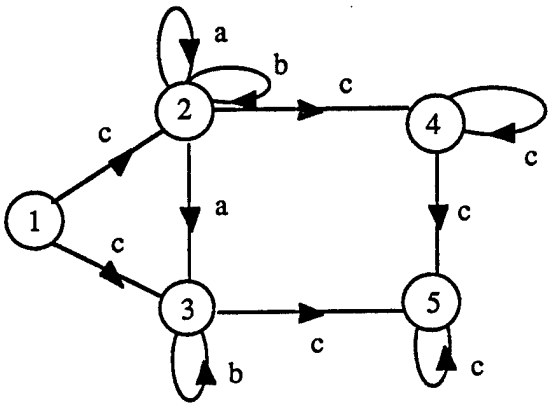
A transition $q \xrightarrow{a} q'$ in T may be either a transition of $T \mid Q_i$ for some i or a transition from Q_i to Q_j where $i \neq j$. In the first case $q, q' \in Q_i$ and state q" is equivalent to q modulo h iff $q'' \in Q_i$: we certainly have then for all such $q'' \in Q_i$ a transition labeled by a, with q" as origin and terminating in Q_i for otherwise

$L(S \mid Q_i, q'', Q_i)$ would be different from $\lambda(T \mid Q_i)^*$.

In the second case we have $q, q'' \in Q_i$ and $q' \in Q_j$, where $j \neq i$ and the condition of the theorem imply the existence of a transition labeled by a with q" as origin and terminating in Q_j. □

This theorem gives us a procedure to find all the MR quotients of a given TS. We look for all the strongly right monoïdal subsets of Q : clearly every subset reduced to one state is strongly right monoïdal and we can form partitions. For each of them we check the MR condition.

Exemple



S_1

In order to compute all the MR quotients of S_1 we need look at all the MR partition

- certainly the trivial one $\{1, 2, 3, 4, 5\} = \{1\} \cup \{2\} \cup \{3\} \cup \{4\} \cup \{5\}$ is MR (this is a general phenomen)

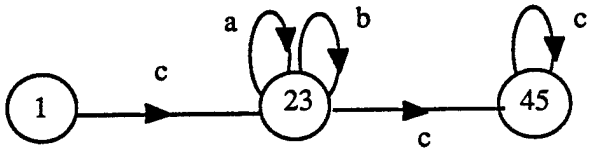
- we can find only 2 strongly right monoidal subsets of Q

{2,3} is such that L(S | {2,3}, 2, {2,3}) = L(S | {2,3}, 3, {2,3} = $(a \cup b)^*$

{4,5} is such that L(S | {4,5}, 4, {4,5}) = L(S | {4,5}, 5, {4,5}) = c^*

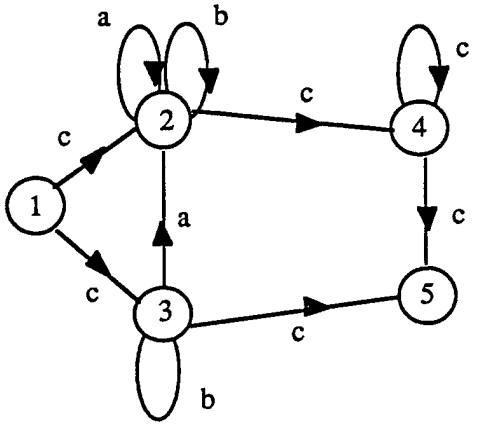
And we check that $\{1\} \cup \{2,3\} \cup \{4,5\}$ is an MR partition : we just have to check that there exists a transition from 3 to {4,5} labeled by c since there exists a transition from 2 to {4,5} labeled by c.

Then we can form the MR quotient S_2



There are no more MR partitions.

A slight alteration of S_1 gives us a MR minimal transition system



S_3

for now $\{4,5\}$ is not strongly right monoidal.

Thus the only partition which could be MR is

$$\{1\} \cup \{2,3\} \cup \{4\} \cup \{5\}$$

but it is not since we have $2 \xrightarrow{c} 4$ and for the partition to be MR there should be a transition $3 \xrightarrow{c} 4$. Thus S_3 has no MR quotient different from itself and is MR minimal.

II. Functional equivalence

A family H of morphism is said to be anti-Church Rosser (abbreviated ACR) iff it satisfies the properties

- H contains the identity
- H is closed under composition
- for all S_1, S_2, S_3 and morphisms h_1 and h_2 in H such that $S_3 = h_1(S_1) = h_2(S_2)$ there exists a transition system S and two H-morphisms h'_1 and h'_2 satisfying

$h_1(S) = S_1$ and $h_2(S) = S_2$.

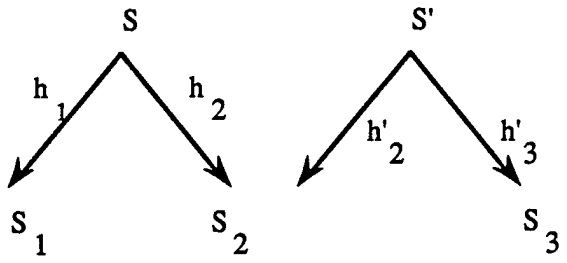
Property II.1 *If the family H is ACR the relation between transition systems defined by*

$$S_1 \sim_H S_2 \Leftrightarrow \exists\, S\, \exists\, h_1, h_2 \in H \quad S_1 = h_1(S) \text{ and } S_2 = h_2(S)$$

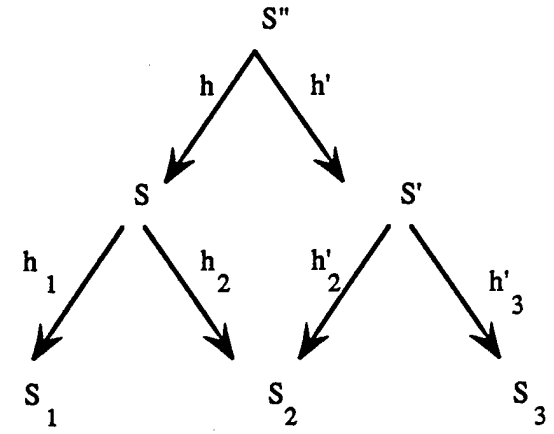
in an equivalence relation.

Proof : Indeed we just have to check the transitivity of this relation.

If $S_1 \sim_H S_2$ and $S_2 \sim_H S_3$ we are in the situation described by the following diagram



and using the ACR property we can complete this diagram to get



Since $h_o h_1$ and $h'_o h'_3$ are in H which is closed by composition this diagram proves that $S_1 \sim S_3$

Property II.2 *The family Mor of all morphisms is ACR.*

Proof We consider the amalgamated product $S_1 \otimes S_2$ of two transitions systems.

This product is the transition system S given by

$$Q = Q_1 \times Q_2$$

$$T = T_1 \otimes T_2 = \{(q_1, q_2) \xrightarrow{a} (q'_1, q'_2) \mid q_1 \xrightarrow{a} q'_1) \in T_1 \text{ and } (q_2 \xrightarrow{a} q'_2) \in T_2\}$$

The computations of $S_1 \otimes S_2$ are amalgamated products of computations of S_1 and S_2 ie

$(q_0\ \bar{q}_0) \xrightarrow{a_1} (q_1, \bar{q}_1) \xrightarrow{a_2} \ldots \xrightarrow{a_n} (q_n,, \bar{q}_n)$ is a computation of $S_1 \otimes S_2$ iff two projections

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} (q_n, \bar{q}_n \text{ and } \bar{q}_0 \xrightarrow{a_1} \bar{q}_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n$$

are computations of S_1 and S_2 respectively.

If $S_3 = h_1(S_1) = h_2(S_2)$ is a common quotient of S_1 and S_2 and π_1, π_2 are the two projections of $Q_1 \times Q_2$ onto Q_1 and Q_2 one has $S_1 = \pi_1(S_1 \otimes S_2)$ and $S_2 = \pi_2(S_1 \otimes S_2)$.

Clearly $\pi_1(T_1 \otimes T_2) \subseteq T_1$.

The reverse inclusion comes from the easy fact that if

$h_1(S_1) = h_2(S_2)$ then by property I.3 $\lambda(T_1) = \lambda(T_2)$.

Thus for every $(q_1 \xrightarrow{a} q'_1) \in T_1$ there exists $(q_2 \xrightarrow{a} q'_2) \in T_2$ with the same label a and $T_1 \otimes T_2$ contains $(q_1, q_2) \xrightarrow{a} (q'_1, q'_2)$ where first projection is $q_1 \xrightarrow{a} q'_1$. □

The two equivalences $\equiv_{Mor}$ and $\sim_{Mor}$ corresponding to the family of all morphisms are identical ie one has

$$S_1 \equiv_{Mor} S_2 \Leftrightarrow \lambda(T_1) = \lambda(T_2) \Leftrightarrow S_1 \sim_{Mor} S_2$$

Property II.2 *The family F of functional morphismls is ACR.*

We prove the more precise property.

Property II.3 *If there exists two functional morphisms* h_2 *and* h_2 *such that* $h_1(S_1) = h_2(S_2)$ *then the two projections of* $S_1 \otimes S_2$ *onto* S_1 *and* S_2 *are functional.*

Proof Assume $h_1(S_1) = h_2(S_2)$ with $h_1, h_2 \in F$.

Every computation in $S_3 = h_1(S_1) = h_2(S_2)$ is the image under h_1 (resp. h_2) of a computation in S_1 (resp. S_2).Thus if $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n$ is a computation in S_3 there exist one computation in S_1, namely $\bar{q}_1 \xrightarrow{a} \bar{q}_2 \rightarrow \ldots \xrightarrow{a_n} \bar{q}_n$

and one computation in S_2, namely,

$\bar{\bar{q}}_0 \xrightarrow{a_1} \bar{\bar{q}}_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} \bar{\bar{q}}_n$ which satisfy

$\forall i \in \{0,\ldots,n\} : h_1(\bar{q}_i) = q_i$ and $h_2(\bar{\bar{q}}_i) = q_i$

The amalgamated product of these two computations, namely,

$(\bar{q}_0, \bar{\bar{q}}) \xrightarrow{a} (q_1, \bar{\bar{q}}) \xrightarrow{a_2} \ldots \xrightarrow{a_n} (\bar{q}_n, \bar{\bar{q}}_n)$

is a computation in $S_1 \otimes S_2$.

Thus for every computation c_1 in S_1 one can find a computation c_2 in S_2 such that $h_2(c_2) = h_1(c_1)$ and $c_1 \otimes c_2$ is a computation in $S_1 \otimes S_2$ clearly satisfying $\pi_1(c_1 \otimes c_2) = c_1$.

We have proved property II.3 and property II.2.

Definition The equivalence $\sim_F$ associated to the ACR family of functional morphisms is called the functional equivalence.

Theorem II.1 *Two transitions systems are functionally equivalent iff their global languages are identical.*

Proof $S_1 \sim_H S_2$ implies the existence of two functional morphisms L_1 and h_2 and a transition system S such that $S_1 = h_1(S)$ and $S_2 = h_2(S)$. Since h_1 and h_2 are functional we have

$$\mathcal{L}(S_1) = \mathcal{L}(S) = \mathcal{L}(S_2)$$

Conversely we assume that $\mathcal{L}(S_1) = \mathcal{L}(S_2)$: we prove that $\mathcal{L}(S_1) = \mathcal{L}(S_2) = \mathcal{L}(S_1 \otimes S_2)$ and that the two projections of $S_1 \otimes S_2$ onto S_1 and S_2 are functional.

Consider a word $u = a_1 \ldots a_n$ in $\mathcal{L}(S_1)$ and a computation $c = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n$ of S_1 such that $\lambda(c) = u$. Since u belongs to $\mathcal{L}(S_2)$ there exists a computation $\bar{c} = \bar{q}_0 \xrightarrow{a_1} \bar{q}_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} \bar{q}_n$ of S_2 such that $\lambda(\bar{c}) = u$.

The amalgamated product of c and $\bar{c}$ is a computation of $S_1 \otimes S_2$ such that $\lambda(c \otimes \bar{c}) = u$, $\pi_1(c \otimes \bar{c}) = c$, $\pi_2(c \otimes \bar{c}) = \bar{c}$. Thus $\mathcal{L}(S_1) \subseteq \mathcal{L}(S_2)$ implies $\mathcal{L}(S_1) = \mathcal{L}(S_1 \otimes S_2)$ and the functionality of π_1. The proof follows immediately. □

Functional partitions

In this paragraph we characterize the partitions corresponding to functional morphisms, called functional partitions or F-partitions. We first remark that

Property III.4

Each component of a F-partition is a monoidal subset of Q.

Proof

It is immediate from the proof of property I.7 and the fact that the morphism h is functional. This implies that $Comp(h(S) \mid \{i\}, \{i\}, \{i\}) = h(Comp(S \mid Q_i, Q_i, Q_i)$ whence $\mathcal{L}(h(S) \mid \{i\}) = \lambda(T \mid Q_i)^* = \mathcal{L}(S \mid Q_i)$. □

Definition II. Let Q' be a monoïdal subset of Q such that $\lambda(T \mid Q') = A'$ and $\mathcal{L}(S \mid Q_i) = A^*_i$. We say that a pair (Q'_1, Q'_2) of subsets of Q' is a complete input-output system for Q' iff

$$A'^* = \cup \{L(S \mid Q', q'_1, q'_2) \mid q'_1 \in Q'_1, q'_2 \in Q'_2\}$$

Theorem II.2 The following conditions are necessary and sufficient for a partition $Q = Q_1 \cup \ldots \cup Q_h$ to be functional

- for all $i \in [h]$ Q_i is monoïdal
- for all $i,j \in [h]$ and all $a \in A$ there exists two subsets of Q_i denoted $Q_{i,j,a}$ and $Q_{i,a,j}$ which satisfy the following conditions :
  - $Q_{i,j,a} = \emptyset$ iff $\{t \in T \mid \alpha(t) \in Q_j, \beta(t) \in Q_i, \lambda(t) = a\} = \emptyset$
  - $Q_{i,a,j} = \emptyset$ iff $\{t \in T \mid \alpha(t) \in Q_i, \beta(t) \in Q_j, \lambda(t) = a\} = \emptyset$
  - for all i,j,a,

$\forall\, q_j \in Q_{j,a,i}\ \forall\, q_i\ Q\in\ Q_{i,j,a}\ (q_j \xrightarrow{a} q_i) \in T$

$(Q_i, Q_{i,a,j})$ and $(Q_{i,j,a}, Q_i)$ are complete input-output systems of Q_i if $Q_{i,a,j}$ (resp. $Q_{i,j,a}$) is not empty

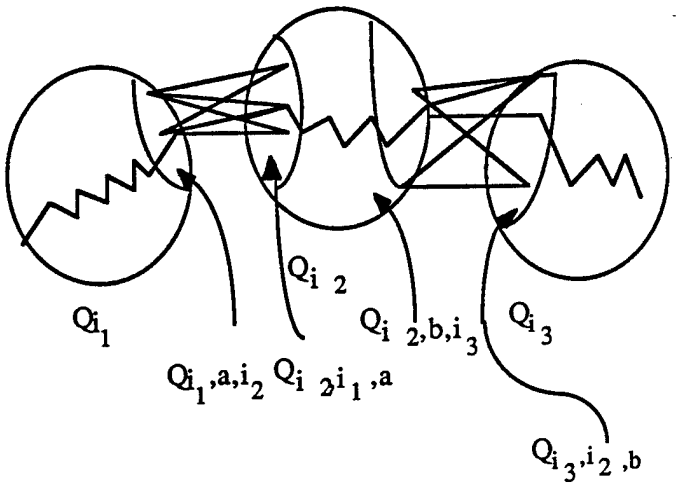
- for all $i,j,\ell \in [h]$, $a,b \in A$ if

$Q_{i,j,a}$ and $Q_{i,b,\ell}$ are non empty

$(Q_{i,j,a}, Q_{i,b,\ell})$ is a complete input-output system of Q_i

Proof These conditions are exactly the necessary conditions so that we can make a computation of S from succession of computations in $S \mid Q_{i_1}$ followed by a transition from Q_{i_1} to Q_{i_2} then a computation in $S \mid Q_{i_2}$ followed by a transition from Q_{i_2} to Q_{i_3} and so on.

Intuitively the situation is described by the following figure



We have only expressed that there are enough transitions between the components of a partition : for all word f in $A^*_{i_2}$ there exists a computation of $S \mid Q_{i_2}$ from one state in $Q_{i_2,i_1,a}$ which can receive a transition labeled by a coming from q_{i_1} to the origin of a transition labeled by b going to Q_{i_z}.

Proof A partition satisfying the conditions of theorem II.2 is an F-partition :

- a computation in h(S) where h maps Q onto [h] can be factorized as follows

$q_{i_1} \xrightarrow{f_i(1)} q_{j_2} \xrightarrow{f_i(2)} q_{j_3} \rightarrow \ldots \xrightarrow{f_i(m)} q_{j_m}$ of $S \mid Q_j$ such that all the q_j belong to Q_j.

The definition of complete input-output systems implies that we can always take (q_{j_1}, q_{j_m}) in $(\overline{Q}_i, \bar{\bar{Q}}_i)$ if $(\overline{Q}_i, \bar{\bar{Q}}_i)$ is a complete input-output system for Q_i.

Then we can use the condition of theorem II.2 to find a computation in S which is mapped by h onto a given computation c' of h(S). If the computation c' contains as a factor

$\ldots i_1 (\xrightarrow{a_1} i_2 \xrightarrow{f_2} i_2 \xrightarrow{a_2} i_3 \ldots$

we shall replace this factor by

$\ldots (\xrightarrow{a_1} q_{i_2}, 1 \xrightarrow{f_2(1)} q_{i_2,2} \xrightarrow{f_2(2)} \ldots \xrightarrow{f_2(m)} q_{i_2,m} \xrightarrow{a_2}$ where $(q_{i_2,1}, q_{i_2,m}) \in (Q_{i_2,i_1,a_1}, Q_{i_2,a_2,i_3})$ which is a complete input-output system for Q_{i_2}.

The leftmost and right most factors will be replaced by

$q_{i_1,1} \xrightarrow{f_i(1)} \ldots \xrightarrow{f_i(m)} q_{i_1,m}$ where $q_{i_1,m} \in Q_{i_1,a_1,i_2}$

$q_{i_m,1} \xrightarrow{f_n(1)} \ldots \xrightarrow{f_n(m)} q_{i_n,m}$ where $q_{i_n,1} \in Q_{i_n,i_{n-1},a_n}$

We have clearly build a computation c in S such that $h(c) = c'$.

The same argument shows the necessity of the condition : assume that $Q = Q_1 \cup ... \cup Q_k$ is an F-partition and consider $i, j, l \in [k]$, $a, b \in A$ such that $i \neq j$ and $i \neq \ell$... if $Q_{i,a,j} \neq \emptyset$ and $Q_{i,b,l} \neq \emptyset$ we have in h(S) the computation c' :

$$j \xrightarrow{a} i \xrightarrow{f} i \xrightarrow{b} l$$

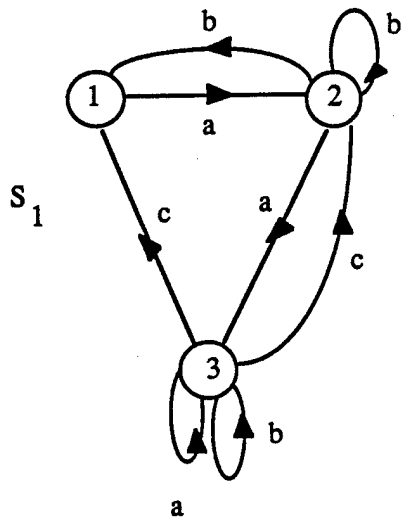
If $(Q_{i,j,a}, Q_{i,b,l})$ is not a complete input output system for Q_i then we can find an $f \in \mathcal{L}(h(S) \mid \{i\})$ such that for all computations $q_{i_1} \xrightarrow{f} q_{i_n}$ of $S \mid Q_i$ either $q_{i_1} \notin Q_{i,j,a}$ or $q_{i_n} \notin Q_{i,b,l}$.

Thus there cannot exist a computation in S

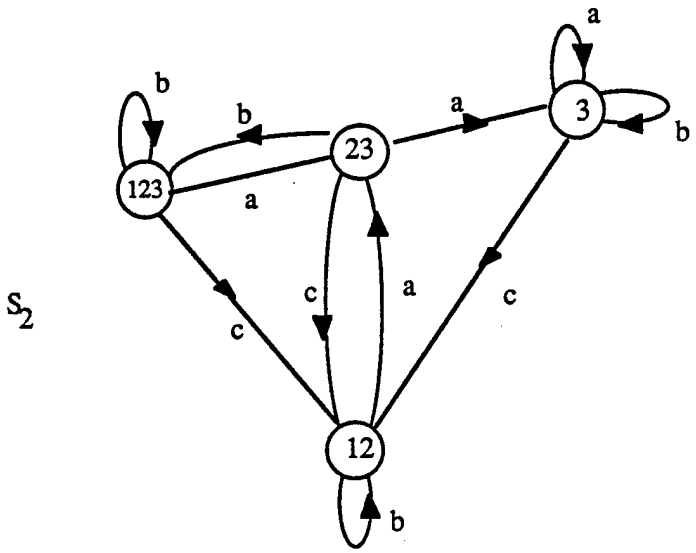
$$q_j \xrightarrow{a} q_{i_1} \xrightarrow{f} q_{i_n} \xrightarrow{b} q_l$$

which is mapped by h on c'. □

Exemple The following system S_1 has no F-quotient.



No subset of {1, 2, 3} with more than one element is monoidal. Let us compute the deterministic equivalent S_2 of S_1. The deterministic equivalent S_2 of S_1 is surely functionally equivalent to S_1.



The system S_2 has an F-quotient (and one only).

The only monoïdal subset of {12, 123, 23, 3} with two elements is

{23, 3} but the corresponding partition

{12} ∪ {23, 3} ∪ {123} is not an F-partition for we can only take

$Q_{\{23,3\},\ \{12\},\ a} = \{23\}$

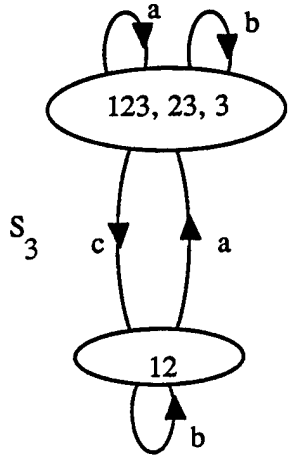
and $L(S_2 \mid \{23, 3\}, 23, \{23, 3\}) = a\ (a \cup b)^* \neq (a \cup b)^*$

The only monoidal subset with 3 elements is {123, 23, 3} and the partition {12} ∪ {123, 23, 3} is an F-partition for we can take

$Q_{\{123,\ 23,\ 3\},\ \{12\},\ a} = \{23\}$ and $Q_{\{123,\ 232,\ 3\},\ c,\ \{12\}} = \{123, 23, 3\}$

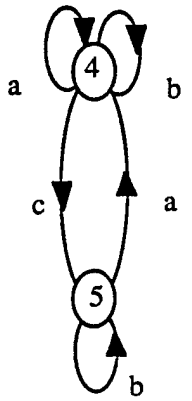
And {23}, {123, 23, 3} is a complete input output system for {123, 23, 3}.

The corresponding F-quotient of S_2 is S_3



S_3

isomorphic to



Since $S_2 \sim_F S_1$ and $S_2 \sim_F S_3$ we have $S_1 \sim_F S_3$. We can check it by computing $S_1 \otimes S_3 = S_4$

Algebra and Communicating Processes

Jos C.M. Baeten*
Department of Software Technology,
Centre for Mathematics and Computer Science,
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands.

As an example of the use of algebraic methods in computer science, the theory ACP, dealing with concurrent communicating processes, is described.

1. INTRODUCTION.

Process algebra is the study of concurrent communicating processes in an algebraic framework. As the initiator of this field we consider R. MILNER, with his Calculus of Communicating Systems [M80], which formed the basis for most of the axiom systems in the theory ACP of BERGSTRA & KLOP [BK84, BK85]. The endeavor of process algebra is to treat concurrency theory (the theory of concurrent communicating processes) in an *axiomatic* way, just as for instance the study of mathematical objects as groups or fields starts with an axiomatization of the intended objects. The axiomatic method which concerns us, is algebraic in the sense that we consider structures (also called process algebras by some people) which are models of some set of (mostly) equational axioms; these structures are equipped with several operators. Thus, we use the term *algebra* in the sense of model theory.

There is ample motivation for such an axiomatic-algebraic approach to concurrency theory. The main reason is that there is not one definite notion of *process*. There is a staggering amount of properties which one may or may not attribute to processes, there are dozens of views (*semantics*) which one may have on (a particular kind of) processes, and there are infinitely many models of processes. So an attempt to organize this field of process theories leads very naturally and almost unavoidably to an axiomatic methodology – and a curious consequence is that one has to answer the question "What is a process?" with the seemingly circular answer "A process is something that obeys a certain set of axioms ... for processes". The axiomatic method has proven effective in mathematics and mathematical logic – and in our opinion it has its merits in computer science as wel, if only for its organizing and unifying power.

Next to the organizing role of this set-up with axiom systems, their models and the study of their relations, we have the obvious *computational* aspect. Even more than in mathematics and mathematical logic, in computer science it is *algebra* that counts – the well-known etymology of the word should be convincing enough. For instance, in a system verification the use of transition diagrams may be very illuminating, but especially for larger systems it is evidently desirable to have a formalized mathematical language at our disposal in which specifications, computations, proofs can be given in what is in principle a linear notation (evidenced by [B89]). Only then can we hope to succeed in attempts to mechanize our dealings with the objects of interest. In our case the mathematical language is algebraic, with basic constants, operators to construct larger processes, and equations defining the nature of the processes under consideration. (The format of pure equations is not always enough, though. On occasion, conditional equations and some infinitary proof rules are used.) To be specific: we will always insist on the use of congruences, rather than mere equivalences in the construction of process algebras; this in order to preserve the purely algebraic format.

---

* Partial support received by ESPRIT contract 432, A formal integrated approach to industrial software development (METEOR), and RACE contract 1046, Specification and Programming Environment for Communication Software (SPECS).

A further advantage of the use of the axiomatic-algebraic method is that the entire apparatus of mathematical logic and the theory of abstract data types is at our disposal. One can study extensions of axiom systems, homomorphisms of the corresponding process algebras. One can formulate exact statements as to the relative expressibility of some process operators (non-definability results).

Of course, the present axiomatizations for concurrency theory do not cover the entire spectrum of interest. Several aspects of processes are as yet not well treated in the algebraic framework. The most notable examples concern the real-time behaviour of processes, and what is called *true concurrency* (non-interleaving semantics). Algebraic theories for these aspects are under development at the moment (see e.g. VAN GLABBEEK & VAANDRAGER [GV87]).

In our view, process algebra can be seen as a worthy descendant of 'classical' automata theory as it originated three or four decades ago. The crucial difference is that nowadays one is interested not merely in the execution traces (or language) of one automaton, but in the behaviour of systems of communicating automata. As Milner and also HOARE [H85] have discovered, it is then for several purposes no longer sufficient to abstract the behaviour of a process to a language of execution traces. Instead, one has to work with a more discriminating process semantics, in which also the timing of choices of a system component is taken into account. Mathematically, this difference is very sharply expressed in the equation $x\cdot(y + z) = x\cdot y + x\cdot z$, where + denotes choice and $\cdot$ is sequential composition; x,y,z are processes. If one is interested in languages of execution traces (trace semantics), this equations holds; but in process algebra it will in general not hold. Nevertheless, process algebra retains the option of adding the equation and studying its effect. In fact, one goal of process algebra is to form a uniform framework in which several different process semantics can be compared and related. One can call this *comparative concurrency semantics*.

We bring structure in our theory of process algebra by *modularization*, i.e. we start from a minimal theory (containing only the operators $+,\cdot$), and then we add new features one at at time. This allows us to study features in isolation, and to combine the modules of the theory in different ways.

In the following, we give a survey of the theory ACP (Algebra of Communicating Processes) as introduced in [BK84].

2. BASIC PROCESS ALGEBRA.

Process algebra starts with a given set A of **atomic actions** $a,b,c,....$ These actions are taken to be indivisible and to have no duration. When we describe a certain application, we will have to be specify what are the atomic actions involved. Thus, the set A will form a *parameter* of our theory. Each atomic action is a constant in the theory. Actions can be combined into composite processes by the operators + and $\cdot$. + is **alternative composition**, choice or sum, and $\cdot$ is **sequential composition** or product. Thus, $(a+b)\cdot c$ is the process that first chooses between executing a or b, next executes c and then terminates. Since time has a direction, product is not commutative; but sum is, and in fact it is stipulated that the options (summands) possible in some state of the process form a *set*. Formally, we will require that all processes $x,y,...$ satisfy the axioms in table 1.

$x + y = y + x$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$(x + y)\cdot z = x\cdot z + y\cdot z$	A4
$(x\cdot y)\cdot z = x\cdot(y\cdot z)$	A5

Table 1. BPA.

We often leave out brackets and the product sign, as in regular algebra. Product will bind more strongly than other operators, sum will bind more weakly. Thus, $xy + z$ means $(x\cdot y) + z$. The theory in table 1 is called Basic Process Algebra or BPA.

We do not include an axiom $x(y + z) = xy + xz$ because the moment of choice in both processes is different, and this difference is important in many applications. For instance, a game of Russian roulette could be described by spin·click + spin·bang, but not by spin·(click + bang).

3. TERMINATION.

Let us again consider sequential composition of processes. If in process $x \cdot y$ component x has performed all its actions, and can do nothing more, it has terminated successfully and process y starts. But if process x consists of a number of concurrently operating components, that at some point are all waiting for a communication from another component, then x also cannot perform any more actions, but in such a situation we do not want that y start. In the second case, we say x has terminated unsuccessfully, is in a state of **deadlock**, and no action is possible any more. Thus, we want to distinguish between successful and unsuccessful termination. We use the constant δ for unsuccessful termination. The laws for this constant are in table 2.

$x + \delta = x$	A6
$\delta \cdot x = \delta$	A7

Table 2. Deadlock.

Now we can give a more formal argument for rejection of the law $x(y + z) = xy + xz$: a consequence is $ab = a(b + \delta) = ab + a\delta$, and this means that a process with deadlock possibility is equal to one without. In most applications, it is important to model deadlock behaviour, so this is an unwanted identification.

It sometimes has advantages to also include a special constant for successful termination. For this purpose, the **empty process** ε is often used, with laws $\varepsilon x = x = x\varepsilon$.

4. INTERLEAVING.

If we look at the **parallel composition** $x \| y$ of processes x and y from the outside, we will see that the atomic actions of x and y are *interleaved* or merged in time (since we assume they have no duration). Thus, at each point in time, the next action will either come from x or from y. In order to get a finite axiomatisation for the parallel composition or **merge**, we will use an auxiliary operator $\lfloor\!\lfloor$ (left-merge). $x \lfloor\!\lfloor y$ is just like $x \| y$, but with the restriction that the first step comes from x.

$x \| y = x \lfloor\!\lfloor y + y \lfloor\!\lfloor x$	M1
$a \lfloor\!\lfloor x = ax$	M2
$ax \lfloor\!\lfloor y = a(x \| y)$	M3
$(x + y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$	M4

Table 3. Interleaving.

The theory with constants A, operators $+, \cdot, \|, \lfloor\!\lfloor$ and axioms in tables 1 and 3 is called PA. Axioms M2 and M3 are actually *axiom schemes*: we have such an axiom for each $a \in A \cup \{\delta\}$. With the axioms of PA, we can eliminate the operators $\|, \lfloor\!\lfloor$ from all closed terms. In fact, this elimination takes the form of a *term rewrite system*. Thus, merge becomes a defined operator on closed terms (but not on infinite processes, defined by means of recursive equations).

5. COMMUNICATION.

Parallel composition between processes is not interesting without some form of communication. For this reason, we extend the merge operator of section 4 to include the possibilities for communication. First, we need to say which atomic actions can communicate, which actions are communication partners. For this reason, we assume we have a **communication function** γ given on the set of atomic actions A. This is a partial binary function on A; if $\gamma(a,b) = c$, we say that a and b communicate, and the result of the communication is c; if $\gamma(a,b)$ is undefined, we say that a and b do not communicate. We

do not restrict ourselves beforehand to binary communication only, but will require that γ is commutative and associative, i.e. for all $a,b,c \in A$ we have

$$\gamma(a,b) = \gamma(b,a)$$
$$\gamma(a,\gamma(b,c)) = \gamma(\gamma(a,b),c),$$

and either side of these equations is defined exactly when the other side is. Now, the parameters of our theory are A and γ.

In order to incorporate the possibility for communication in the merge operator, we use an additional auxiliary operator $|$ (communication merge). Now, $x \mid y$ is just like $x \| y$, but with the restriction that the first step is a communication step between x and y. In table 4, $a,b \in A \cup \{\delta\}$, and x,y,z are arbitrary processes.

$a \mid b = \gamma(a,b)$	if $\gamma(a,b)$ is defined	CF1
$a \mid b = \delta$	otherwise	CF2
$x \| y = x \lfloor\!\lfloor y + y \lfloor\!\lfloor x + x \mid y$		CM1
$a \lfloor\!\lfloor x = ax$		CM2
$ax \lfloor\!\lfloor y = a(x \| y)$		CM3
$(x + y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$		CM4
$ax \mid b = (a \mid b)x$		CM5
$a \mid bx = (a \mid b)x$		CM6
$ax \mid by = (a \mid b)(x \| y)$		CM7
$(x+y) \mid z = x \mid z + y \mid z$		CM8
$x \mid (y+z) = x \mid y + x \mid z$		CM9

Table 4. Merge with communication.

6. ENCAPSULATION.

In communicating systems, we often want that communication partners should communicate, and not occur by themselves. In order to block unwanted occurrences of such actions, we need the **encapsulation operators** ∂_H. Here, H is a set of atomic actions ($H \subseteq A$), and ∂_H will block all actions from H, by renaming them into δ.

$\partial_H(a) = \delta$	if $a \in H$	D1
$\partial_H(a) = a$	otherwise	D2
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$		D3
$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$		D4

Table 5. Encapsulation.

The axioms in table 1,2,4 and 5 together constitute the axiom system ACP (Algebra of Communicating Processes) of BERGSTRA & KLOP [BK84]. Typically, a system of communicating processes $x_1,\ldots,x_n$ is represented in ACP by the expression $\partial_H(x_1 \| \ldots \| x_n)$, where H will contain all communication 'halves' occurring in the parallel composition.

This language (with some extra defined operators) has been used extensively in [1] in system specification. In order to do system *verification*, it is necessary to tackle the issue of *abstraction* as in [BK85].

REFERENCES.

[B89] J.C.M. BAETEN (ed.), *Applications of process algebra,* CWI Monograph 8, North-Holland, Amsterdam 1989. To appear.

[BK84] J.A. BERGSTRA & J.W. KLOP, *Process algebra for synchronous communication,* Inf. & Control 60 (1/3), 1984, pp. 109-137.

[BK85] J.A. BERGSTRA & J.W. KLOP, *Algebra of communicating processes with abstraction,* Theor. Comp. Sci. 37 (1), 1985, pp. 77-121.

[GV87] R.J. VAN GLABBEEK & F.W. VAANDRAGER, *Petri net models for algebraic theories of concurrency,* in: Proc. PARLE II (J.W. de Bakker, A.J. Nijman & P.C. Treleaven, eds.), Springer LNCS 259, 1987, pp. 224-242.

[H85] C.A.R. HOARE, *Communicating sequential processes,* Prentice-Hall, 1985.

[M80] R. MILNER, *A calculus of communicating systems,* Springer LNCS 92, 1980.

Parametrized Process Categories

Extended Abstract

Roger F. Crew*
Computer Science Department
Stanford University

January 1989

Abstract

In any model of computation that distinguishes concurrency from nondeterminism, it is useful to be able to independently associate structure with each of these notions. To this end, we develop a categorical definition of process based on the pomset model parametrized on the choices of temporal structure and notion of nondeterminism.

1 Introduction

Partial order based models of concurrency [Gre75, Gra81, NPW81, MS80, Pra82, PW84] vary in their treatment of nondeterminism. Emulating formal language and trace theory, the labeled partial order (also called a partially ordered multiset — *pomset*) model of Grabowski [Gra81] and Pratt [Pra82] defines a process to be a set of *behaviours*, a behaviour being a set of events, a collection of timing constraints on the events, and a labeling associating each event with an action from another set (the *alphabet*). [1] In the original version, a behaviour is simply a pomset, that is, the timing constraints are given by a partial order.

In this scheme, the notion of behaviour is strictly a deterministic one; all events of the behaviour must occur. Real nondeterminism (as opposed to the spurious nondeterminism introduced by the observation/interleaving of concurrent events — a distinction that makes sense from the "true concurrency" point of view) only appears at the upper level where a choice is made concerning which of the alternative behaviours of the process is to be executed. This "disjunctive normal

*Based partially on work supported by an NSF Graduate Fellowship

[1] The Petri net literature refers to events as *event occurrences* and to actions as *events*.

form" representation for processes has the advantage of allowing us to treat the nondeterministic and concurrent aspects somewhat independently. [2]

In other work, we (with Casley, Meseguer and Pratt [CCMP89]) generalized the notion of behaviour to include other forms of temporal constraint besides that given by a partial order. Alternative timing schemes included preorders, prosset orders (as defined in [GP87]), and premetric spaces [Law73], to name a few. Particularly important to us was the ability to provide rather clean categorical definitions for many of the operations on behaviours described in [Gis84, Pra86] (e.g., concurrence, concatenation, orthocurrence, pomset-definables) which did not depend on the underlying temporal structure being that of a partial order.

We now want to accomplish the same for full-fledged processes that include actual nondeterminism. How should the notion of process change as we change the timing schemes for events? As with the behaviours it is important that we provide suitably abstract definitions for the major *process* operations described in [Gis84, Pra86] if we expect to use them in specifications involving notions of temporal constraint more detailed than that of partial orders.

As with timing schemes, other notions of nondeterminism can also be considered. For example, [BM84] consider notions providing each alternative behaviour with a path count (number of ways of producing this alternative) or a predicate governing the occurence of each alternative. These possibilities should also be included in our process model.

In this paper we achieve both of these goals with our model, in effect composing the two categories representing the disjunctive and conjunctive structure respectively. We can also introduce structure between the alternatives as well; the set of alternatives becomes a category including partial stages in a computation as well as completed behaviours.

The construction starts with an appropriate category of behaviours $\mathcal{B}$ (the conjunctive structure). If we consider behaviours to represent the actions taken by a system or a component of a system, a behaviour morphism is best viewed for our purposes as mapping the actions/events of subcomponents into those of larger components which contain them. Each event of the subcomponent appears somewhere in the larger component; the morphism tells us where. One useful consequence of this particular interpretation is that given a diagram of component behaviours, we can derive the full system behaviour by the simple expedient of taking a colimit.

When considering the nondeterministic aspect, we notice that this relationship is reversed; each alternative available to the component implies a particular alternative in the *sub*component. This reversal is a consequence of the disjunctive nature of processes versus the conjunctive nature of behaviours. We then take a process P to be a set A_P indexing the available alternatives, together with a function $P : A_P \to \mathcal{B}$ identifying the behaviours.

[2] Contrast this with other models that incorporate the nondeterministic and concurrent aspects into a single one-level structure (e.g, event-structures).

A_P can also be thought of as a discrete category with P a functor.

A process morphism $f : P \to Q$ then consists of a functor $A_f : A_Q \to A_P$ (note the reversal) and a natural transformation $\xi_f : PA_f \to Q$. (there are variants of this construction taking into account the alphabets of the behaviours of $\mathcal{B}$ which will discussed).

The disjunctive structure is handled similarly, wherein we have a suitable category $\mathcal{E}$ of e.g., predicates or multiplicities. Our notion of process will then also include a (contra)functor $P' : A_P \to \mathcal{E}$, while process morphisms now include a natural transformation $\zeta_f : Q' \to P'A_f$. We leave open both the choice of behaviour notion $\mathcal{B}$ and the choice of nondeterminism notion $\mathcal{E}$.

As with the behaviours we get, assuming $\mathcal{B}$ and $\mathcal{E}$ are sufficiently well behaved, generalizations of the various process operations defined for many pomset processes described in [Pra86] and [Gis84] by taking them to be appropriate limits or colimits (e.g., union is a product, concurrence is a coproduct). Fixpoint constructions [AK79, PS78] can also work. We again, as with behaviours, obtain a straightforward notion of system composition from taking colimits, albeit a somewhat different one from those previously proposed (i.e., Y-section [Gra81], utilization [Pra86] and fusion [GP87]).

References

[AK79] J. Adámek and V. Koubek. Least fixed point of a functor. *Journal of Computer and System Sciences*, 19:163–178, 1979.

[BM84] D. B. Benson and M. G. Main. Functional behavior of nondeterministic and concurrent programs. *Inform. & Control*, 62:144–189, 1984.

[CCMP89] R. Casley, R. F. Crew, J. Meseguer, and V.R. Pratt. Dynamic structures. *(to appear)*, 1989.

[Gis84] J. Gischer. *Partial Orders and the Axiomatic Theory of Shuffle.* PhD thesis, Computer Science Dept., Stanford University, December 1984.

[GP87] H. Gaifman and V.R. Pratt. Partial order models of concurrency and the computation of functions. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 72–85, Ithaca, NY, June 1987.

[Gra81] J. Grabowski. On partial languages. *Fundamenta Informaticae*, IV.2:427–498, 1981.

[Gre75] I. Greif. *Semantics of Communicating Parallel Processes.* PhD thesis, Project MAC report TR-154, MIT, 1975.

[Law73] W. Lawvere. Metric spaces, generalized logic, and closed categories. In *Rendiconti del Seminario Matematico e Fisico di Milano, XLIII.* Tipografia Fusi, Pavia, 1973.

[MS80] U. Montanari and C. Simonelli. On distinguishing between concurrency and nondeterminism. In *Proc. Ecole de Printemps on Concurrency and Petri Nets*, Colleville, 1980.

[NPW81] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures, and domains, part i. *Theoretical Computer Science*, 13, 1981.

[Pra82] V.R. Pratt. On the composition of processes. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, January 1982.

[Pra86] V.R. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, February 1986.

[PS78] G.D. Plotkin and M.B. Smyth. The category-theoretic solution of recursive domain equations. Technical Report Research Report No. 60, D.A.I., 1978.

[PW84] S.S. Pinter and P. Wolper. A temporal logic to reason about partially ordered computations. In *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, pages 28–37, Vancouver, August 1984.

Equality-Test and If-Then-Else Algebras: Axiomatization and Specification

Don Pigozzi
Iowa State University

Many of the data structures that arise in practice include a Boolean sort together with equality tests for the elements of each non-Boolean sort; in some cases they also include if-then-else operations that select elements of a data domain on the basis of a Boolean test. We find a finite set of conditional equations and a finite set of ordinary equations that axiomatize respectively the classes of equality-test and if-then-else algebras of each appropriate signature. We show that for equality-test algebras conditional specification is as powerful as universal specification, and equational specification is as powerful as universal specification in the presence of the if-then-else operations. We also investigate the power of conditional and equational specifications when the equality tests and if-then-else operations are hidden.

Equality-test Algebras

A signature Σ is an *equality-test signature* it it has a sort *bool* with operation symbols *and*, *or*, *not*, *true*, *false*, and, for each sort $s \neq bool$, an operation symbol $eq_s : s\,s \to bool$. A Σ-algebra $\mathbf{A}$ is an *equality-test algebra* if S contains a sort *bool* such that $\mathbf{A}_{bool}$ is the two-element Boolean algebra, and, for each $s \in S \setminus \{bool\}$, $\Sigma_{s\,s\,bool}$ contains a operation symbol eq_s such that

$$eq_s^{\mathbf{A}}(a,b) = \begin{cases} true, & \text{if } a = b; \\ false, & \text{if } a \neq b. \end{cases}$$

The class of all equality-test Σ-algebras is denoted by ET_Σ; the subscript Σ is omitted when there is no chance of confusion.

A Σ-algebra $\mathbf{A}$ is a *generalized equality-test algebra* if Σ is an equality-test signature and $\mathbf{A}$ satisfies the following set of equations and conditional equations ($\varphi \leq \psi$ stands for the Boolean equation $not\,\varphi\ or\ \psi \approx true$).

(Axget_1) A standard system of equational axioms for Boolean algebras;

for each $s \in S \setminus \{bool\}$:

(Axget_2) $eq_s(x,x) \approx true$;

(Axget_3) $eq_s(x,y) \leq eq_s(y,x)$;

(Axget_4) $eq_s(x,y)\ and\ eq_s(y,z) \leq eq_s(x,z)$;

(Axget_5) $eq_{s_0}(x_0,y_0)\ and \ldots and\ eq_{s_{n-1}}(x_{n-1},y_{n-1}) \leq eq_s(\sigma(x_0,\ldots,x_{n-1}),\sigma(x_0,\ldots,x_{n-1}))$, for each $\sigma \in \Sigma_{w,s}$ with $w = s_0 s_1 \ldots s_{n-1}$;

(Axget_6) $eq_s(x,y) \approx true \to x \approx y$.

This set of axioms is denoted by AXGET_Σ, and the quasivariety of all generalized equality-test Σ-algebras, i.e., the class of models of AXGET_Σ, is denoted by GET_Σ.

The following theorem is the analogue for generalized equality test algebras of the Stone representation theorem (in algebraic form) for Boolean algebras.

GET-Representation Theorem. *Every generalized equality-test algebra is isomorphic to a subalgebra of a Cartesian product of equality-test algebras. More generally, given any set E of equations, every generalized equality-test algebra that satisfies E is isomorphic to a subalgebra of a Cartesian product of equality-test algebras that satisfy E.*

Corollary. GET *is the smallest quasivariety that contains all equality-test algebras. Hence* AXGET *is a base for the conditional equations of* ET.

Corollary. *Let* K *be a subquasivariety of* GET *defined relative to* GET *by any set E of identities. Then the initial algebra of* K *can be represented as the minimal subalgebra of the Cartesian product of all equality-test data structures that satisfy E.*

It follows that an equality-test data structure can be an initial algebra of K iff it is the only equality-test data structure (up to isomorphism) satisfying E. In this case it is clearly also the final algebra of K.

Specification of Equality-Test Algebras

A *data structure* **A** is a heterogeneous algebra that is *minimal* in the sense that it has no proper subalgebras. If there is at least one ground term of each sort, then a data structure may be characterized as a heterogeneous algebra in which each element is denoted by a ground term. (A *ground term* is any term without variables.)

Let Σ be any signature, **A** a Σ-data structure, and Γ a set of first-order Σ-sentences. Γ is an *initial specification* of **A** if **A** is the initial object in the category whose objects are the minimal subalgebras of models of Γ and whose morphisms are homomorphisms. (If Γ is a set of universal sentences, then it is an initial specification in the above sense iff **A** is initial in the category of models of Γ.) Γ is a *final specification* of **A** if **A** is the final (i.e., terminal) object in the category of non-trivial minimal subalgebras of models of Γ. (This particular notion of final specification is due to Bergstra and Tucker [SIAM J. Comput., 12(1983)]). A specification Γ is *complete* if it is at the same time initial and final. A specification is *universal, conditional*, or *equational* if Γ is respectively a set of universal first-order sentences, conditional equations (quasi-equations), or equations.

Let Σ be and equality-test signature. For each quantifier-free Σ-formula φ we define a *bool*-term φ^*, with the same variables as φ, by recursion on the structure of φ. If φ is an s-equation $t \approx r$, then $\varphi^* = eq_s(t,r)$. $(\varphi \wedge \psi)^* = (\varphi^*)$ *and* (ψ^*), $(\varphi \vee \psi)^* = (\varphi^*)$ *or* (ψ^*), and $(\neg\varphi)^* = not(\varphi^*)$. The definition is extended to universal sentences: If φ is a universal sentence, and

$$\forall x_0 \forall x_1 \ldots \forall x_{n-1} \varphi'(x_0, \ldots, x_{n-1})$$

is its prenex normal form with φ' quantifier-free, then $\varphi^* = \varphi'^*$.

φ^* is called the *Boolean transform of* φ. For any set Γ of universal sentences let $E(\Gamma) = \{\, \varphi^* : \varphi \in \Gamma \,\}$.

Theorem 1. *Let Γ be an arbitrary set of universal sentences. The relative subvariety of* GET *defined by $E(\Gamma) \cup$* AXGET *is the smallest quasivariety containing all equality-test algebras that satisfy Γ. Hence $E(\Gamma) \cup$* AXGET *is a base for the conditional equations of the equality-test algebras that satisfy Γ.*

This theorem provides the means for converting any universal initial or final specification of an equality-test algebra into a conditional complete specification.

Corollary. *Let Γ be any set of universal sentences, and let $\mathbf{A}$ be an equality-test data structure. If Γ is either an initial or final specification of $\mathbf{A}$, then $E(\Gamma) \cup \mathrm{AXGET}$ is a conditional complete specification of $\mathbf{A}$.*

Corollary. *Every equality-test data structure that has a finite universal initial specification is computable.*

Conditional Specifications with Hidden Sorts and Operations

We extend the results of the last section to data structures that are not equality-test algebras. A signature Σ' is an *enrichment* of Σ if the sort set S' of Σ' includes the sort set S of Σ and $\Sigma_{w\,s} \subseteq \Sigma'_{w\,s}$ for all $w\,s \in S^* \times S$. A Σ'-algebra $\mathbf{A}'$ is an *enrichment* of a Σ-algebra $\mathbf{A}$ if $\mathbf{A}$ is obtained from $\mathbf{A}'$ by disregarding the additional sorts and operations. In this case $\mathbf{A}$ is called a *reduct* of $\mathbf{A}'$ and is denoted by $\mathbf{A}'|_\Sigma$. A set of Σ'-sentences is an *initial* (*final, complete*) *specification* of a Σ-data structure $\mathbf{A}$ *with hidden sorts and operations* if it is an initial (final, complete) specification of some Σ'-enrichment of $\mathbf{A}$.

Let Σ be an arbitrary signature. The equality-test signature Σ^+ is obtained by enriching Σ with a new Boolean sort *newbool* and a new binary operation eq_s for each sort s of Σ. *newbool* is called the *hidden sort* and the eq_s the *hidden operations* of Σ^+. For an arbitrary Σ-algebra $\mathbf{A}$ the *equality-test enrichment* $\mathbf{A}^+$ of $\mathbf{A}$ is defined in the obvious way.

We have the following extension of Theorem 1.

Theorem 2. *Let Σ be any signature and Γ any set of universal Σ-sentences. Let K be the relative subvariety of GET_{Σ^+} defined by $E(\Gamma) \cup \mathrm{AXGET}_{\Sigma^+}$. Then $\mathsf{K}|_\Sigma$ is the smallest quasivariety containing all Σ-algebras that satisfy Γ. Hence $E(\Gamma) \cup \mathrm{AXGET}_{\Sigma^+}$ is a base for the conditional equations of the models of Γ.*

Corollary. *Let Σ be any signature, Γ any set of universal Σ-sentences, and $\mathbf{A}$ a Σ-data structure. If Γ is an initial specification of $\mathbf{A}$, then $E(\Gamma) \cup \mathrm{AXGET}_{\Sigma^+}$ is a conditional specification of $\mathbf{A}$ with hidden sort and operations.*

In contrast to the case for equality-test data structures, this conditional specification is not in general complete. In fact, if Γ is a universal specification of a data structure $\mathbf{A}$ of arbitrary signature Σ and Γ has no nontrivial models, then $E(\Gamma) \cup \mathrm{AXGET}_{\Sigma^+}$ is a conditional complete specification of $\mathbf{A}$ with hidden sort and operations iff Γ is a universal complete specification of $\mathbf{A}$.

If-Then-Else Algebras

A signature Σ is called an *if-then-else signature* if it is an equality-test signature and, in addition, there exists an operation symbol $[\_,\_,\_]_s : \mathit{bool}\,s\,s \to s$ for each sort $s \neq \mathit{bool}$. A Σ-algebra $\mathbf{A}$ is an *if-then-else algebra* if Σ is an if-then-else signature, $\mathbf{A}$ is an equality-test algebra, and, for each $s \in S \setminus \{\mathit{bool}\}$ and all $b \in \mathbf{A}_{\mathit{bool}}$ and $a_0, a_1 \in \mathbf{A}_s$,

$$[b, a_0, a_1]_s^{\mathbf{A}} = \begin{cases} a_0, & \text{if } b = \mathit{true}; \\ a_1, & \text{otherwise (i.e., } b = \mathit{false}). \end{cases}$$

The class of all if-then-else algebras is denoted by ITE_Σ.

Let Σ be an if-then-else signature, and let AXGITE_Σ be the set of equations obtained from the axioms AXGET_Σ by replacing the one conditional axiom, $eq_s(x,y) \approx \mathit{true} \to x \approx y$, by two equational axioms:

$$[\mathit{true}, x, y]_s \approx x, \quad [eq_s(x,y), x, y]_s \approx y.$$

Any Σ-algebra satisfying AXGITE_Σ is called a *generalized if-then-else algebra.* The variety of generalized if-then-else algebras is denoted by GITE_Σ.

GITE-Representation Theorem. *Every generalized if-then-else algebra is isomorphic to a subalgebra of a Cartesian product of if-then-else algebras. More generally, given any set E of equations, every generalized if-then-else algebra that satisfies E is isomorphic to a subalgebra of a Cartesian product of if-then-else algebras that satisfy E.*

Corollary. GITE *is the smallest quasivariety and also the smallest variety that contains all if-then-else algebras. Hence* AXGITE *is a base for the conditional equations of* ITE.

We also have the following analogues of Theorem 1 and its first corollary; compare Bloom and Tindell [SIAM J. Comput., 12(1983)], Guessarian and Meseguer [SIAM J. Comput., 16(1987)], and Mekler and Nelson [SIAM J. Comput., 16(1987)].

Theorem 3. *Let Γ be an arbitrary set of universal sentences. The subvariety of* GITE *defined by $E(\Gamma) \cup$ AXGITE is the smallest quasivariety and also the smallest variety containing all if-then-else algebras that satisfy Γ. Hence $E(\Gamma) \cup$ AXGITE is a base for the conditional equations of the if-then-else algebras that satisfy Γ.*

Corollary. *Let Γ be any set of universal sentences, and let* **A** *be an if-then-else data structure. If Γ is either an initial or final specification of* **A**, *then $E(\Gamma) \cup$ AXGITE is a conditional complete specification of* **A**.

The if-then-else enrichments Σ^+ of an arbitrary signature Σ and $\mathbf{A}^+$ of an arbitrary Σ-algebra **A** are defined in the obvious way.

Theorem 2 and its corollary do not carry over intact to if-then-else algebras. Their proofs depend on the fact that none of the hidden operations of the equality-test enrichment has a visible sort as target. The best that can be obtained are the following.

Theorem 4. *Let Σ be any signature and Γ any set of universal Σ-sentences. Let* K *be the subvariety of $\mathsf{GITE}_{\Sigma+}$ defined by $E(\Gamma) \cup \mathrm{AXGITE}_{\Sigma+}$. Then the class of subalgebras of $\mathsf{K}|_\Sigma$ is the smallest quasivariety that contains all Σ-algebras that satisfy Γ. Hence $E(\Gamma) \cup \mathrm{AXGITE}^+_\Sigma$ is an equational base for the conditional equations of the models of Γ.*

Corollary. *Let Γ be any signature, Γ any set of universal Σ-sentences, and* **A** *a Σ-data structure. If Γ is an initial specification of* **A**, *then* **A** *is isomorphic to the minimal subalgebra of $\mathbf{B}|_\Sigma$ where* **B** *if the initial algebra of the subvariety of* GITE *defined by $E(\Gamma) \cup \mathrm{AXGET}_{\Sigma+}$.*

Thus a finite universal initial specification of an arbitrary data structure **A** can always be transformed into a finite equational initial specification of a generalized if-then-else-algebra **B** with the property that **A** is a subalgebra of the reduct of **B**. **A** need not be the entire reduct of **B** however, so in general we do not get an equational specification of **A** with hidden sort and operations in the usual sense.

However we do have that, if Γ is a universal specification of a data structure **A** and Γ has no non-trivial models, then $E(\Gamma) \cup \mathrm{AXGITE}_{\Sigma+}$ is a equational complete specification of **A** with hidden sort and operations iff Γ is a universal complete specification of **A**. Compare Bergstra and Tucker [Technical Report IW 156, Math. Cent., Amsterdam, 1980].

On the Algebraic Structure of Petri Net Dynamics

Abstract For AMAST, Iowa City.

David B. Benson
Raju R. Iyer

Computer Science Department
Washington State University
Pullman WA 99164-1210.
dbenson@cs2.wsu.edu

For us, a *petri net* is a Place/Transition Net. The syntax can be a bipartite digraph with constraints and firing rules as in [R82] or categorically motivated monoids as in [MM88a,MM88b]. The operational semantics of petri nets is, roughly speaking, given by finite sequences of markings determined by the graph structure, the constraints and the firing rules. Other related models include [Stk87,Win84,Win87]. Here we say *dynamics* for the operational semantics of petri nets.

The dynamics progresses by a convolution, [Ros88,MB84], which is a commutative monoid on a well supported compact closed structure [Car88]. The algebraic structure of petri nets follows from the general structure theorems in [CW87,Car88] or indeed earlier papers.

Since more than one transition can be enabled in a given marking, there may be several different follower markings from a given marking. Therefore the dynamics is in general nondeterministic.

In this dynamics there is a natural notion of discrete time. Each time step, or tick, corresponds to an attempt to cause all the transitions to fire. But, in this dynamics, even enabled transitions may choose not to fire. So one of the nondeterministic choices is that

no transitions fire on a given tick. Therefore a follower marking may be identical to the predecessor marking. But this identity does not mean no transition has fired, since there are petri nets in which the result of certain firings result in the same marking as before the firing. This is not a defect in the choice of dynamics as the operational syntax. We agree that the intention of a petri net is solely to move indistinguishable tokens from place to place in the petri net. If the act of firing is of particular interest, one may add a distinguished output place to each transition. Each transition adds a token to this distinguished place. In this modified petri net, the number of tokens in each of the additional places records the number of times the transition has fired.

The transitions of the petri net specify nondeterministic functions, roughly from input places to output places, by a delicate transformation from the firing rules to the generators of the nondeterministic functions. The places of the petri net specify nondeterministic distribution functions from the places to the inputs to the transitions. These distribution functions forward tokens to the transitions in all possible ways. The distribution functions are denoted by Δ. Additionally, the places of the petri net specify nondeterministic collection of functions from the outputs of the transitions to the places. The collection functions simply stack all incoming tokens at the place. These functions are denoted by ∇.

Finally, we include an image transition for each place. The image transitions simply copy input to output. This describes the intuition that at any time tick, some or all of the tokens at a place remain at that place. We send such tokens through the image transition associated with the place. For example, a petri net with one place, one transition and set of nondeterministic markings M has a distribution function

$$\Delta : M \longrightarrow M \otimes M$$

which sends the tokens at the place either to the left or the right in all possible ways. The tensor is symmetric monoidal, as in [Mac71,Ben82,Ben87,Ben89]. For n tokens, the result of the distribution is the nondeterministic sum of pairs

$$n\Delta = \sum_{k+p=n} k \otimes p.$$

The collection function from transition outputs to the place has signature

$$\nabla : M \otimes M \longrightarrow M.$$

With k tokens on the left and p tokens on the right, the result of this collection is

$$k \otimes p\nabla = k + p$$

with '+' being the ordinary sum of natural numbers.

The transition of the one place, one transition petri net is associated with the firing function $f : M \longrightarrow M$. There is an image firing function for the place, $p : M \longrightarrow M$. The dynamics, d, of one time tick is the convolution

$$d = \Delta(f \otimes p)\nabla : M \longrightarrow M.$$

These same considerations apply to a petri net with any (usually finite) number of places and transitions. The wiring diagram between places and transitions require some technical care, but causes no conceptual difficulties. Similarly, the structure easily extends to colored tokens and other such variations on the theme.

A category of petri net dynamics has as objects the petri nets and as morphisms nondeterministic functions which preserve the behaviors. Consider the tensor product of two petri nets, this being determined by the tensor product of behavioral convolutions. The full subcategory of petri nets without sources has finite categorical products, these being

the tensor product. The full subcategory of petri nets without sinks has finite categorical coproducts, these being the tensor product. Thus the full subcategory of petri nets without sources or sinks has finite biproducts. This is then the situation of [CW87].

Since the convolutional dynamics d is an endomorphsim, standard methods apply to understanding the iterate over time ticks.

References

[Ben82] Benson, D. B. (1982). "Counting paths: Nondeterminism as Linear Algebra," *IEEE Trans. Software Eng., SE-10, #6 (1984), 785-794.*

[Ben87] Benson, D. B. "The Shuffle Bialgebra," *Proc. 3rd Workshop Math. Found. Programming Language Semantics,* Springer-Verlag LNCS.

[Ben89] Benson, D. B. (1989). "Bialgebras: Some Foundations for Distributed and Concurrent Computation," *Fund. Inform., in press.*

[Car88] Carboni, A. "Matrices, Relations and Group Representations," *Preprint.*

[CW87] Carboni, A. & Walters, R. "Cartesian Bicategories I," *J.Pure Appl. Alg. 49 (1987), 11-32.*

[MB84] Main, M. G. & Benson, D. B. (1984). "Functional Behavior of Nondeterministic and Concurrent Programs," *Inform. & Control, 62:144-189.*

[Mac71]Mac Lane, S. (1971) "Categories for the Working Mathematician," *Graduate Texts in Mathematics, Springer-Verlag.*

[MM88a] Meseguer, J. & Montanari, U. (1988a) "Petri Nets Are Monoids," *SRI-CSL-88-3-January-1988.*

[MM88b] Meseguer, J. & Montanari, U. (1988b) "Petri Nets Are Monoids: A New Algebraic Foundation for Net Theory," *Proc. LICS '88(Edinburgh).*

[R82] Reisig, W. (1982) "Petri Nets- An Introduction," *EATCS, Monograph's on Theoretical Computer Science, Springer Verlag- 1982.*

[Ros88] Rosenberg, I. G. "Algebraic Properties of a General Convolution," *Tech Report, Mathematiques et Statistique, University of Montreal H3C3J7.*

[Stk87] Stark, E. W. (1987) " Concurrent Transition System Semantics of Process Networks," *Proc. ACM Conf. Principles of Prog. Lang., 1987.*

[Win84] Winskel, G. (1984)"Categories of Models for Concurrency," *Workshop on the Semantics of Concurrency, July 1984.*

[Win87] Winskel G. (1987) "Petri nets, algebras, morphisms and compositionality," *Inform. & Comput., 72:197-238.*

Display of graphics and their applications, as exemplified by 2-categories and the Hegelian "taco"

F. William Lawvere[1)]

A graphic monoid M satisfies identically xyx = xy and an application of M is a right M-set. Every left ideal of such an M is also a right ideal, simplifying and structuring the study of the topos of applications. An informal process of displaying pictures of graphics and applications is exemplified, with conjectured use in the organization of knowledge. The Hegelian organization of knowledge is concretely realized in terms of adjoint functors on "any" mathematical category, and is used to give a precise definition of the dimension needed for a display. A central fragment of the Hegelian scheme is revealed as an 8-element graphic, whose suggestive display has reminded some of a taco.

I. INTRODUCTION

By a graphic we will mean any finite category each of whose endomorphism monoids satisfies the identity xyx = xy ; in particular, a graphic monoid is a graphic category with one object. By an application of a graphic category we will mean any right action of it on finite sets (i.e. any contravariant finite-set-valued functor on it). If I is any object of a graphic G , then G(-,I) is a particular application (often called the right regular representation in the case of a monoid) and together these give a full embedding of G into the topos of all applications of G , to which we freely apply the Cayley-Dedekind-Grothendieck-Yoneda lemma. If X is any application of the graphic G , then the "comma" category G/X (whose objects are the elements of X and whose morphisms determine the action via the discrete fibration property of the labelling functor

$G/X \longrightarrow G$) is again a graphic. Thus each particular application X of G provides one way $G' \longrightarrow G$ of expanding the graphic G into a more detailed graphic G' . Even though graphic monoids G play a central role, we must also deal with graphics such as G/X with many objects. Similarly, the category $\overline{G}$ of all retracts of objects of G (which may be constructed either abstractly to have as objects the idempotents of G or concretely as a full subcategory of the category of applications of G ; note that in the former guise it is "a itself" which plays the role of 1_a) will again have many objects - indeed the graphic identity $xyx = xy$ implies $x^2 = x$ so that if G is a monoid then $\overline{G}$ has an object for every element of G (some of those objects may be isomorphic in $\overline{G}$). The interest of $G \hookrightarrow \overline{G}$ is that it induces an equivalence between the associated toposes of applications. We intend to associate with each graphic (by a compelling though not yet well-defined process) a "display" which will reveal much of its structure. We do associate a well-defined distributive lattice which is itself a standard application and which may be considered to consist of refined "dimensions" in that it parameterizes all the ranks in a Hegelian analysis of the topos of all applications; through this distributive lattice there is a well-defined ascending sequence, obtained by the Hegelian process of "resolution of one unity of opposites by the next"; the length of this sequence is the geometrical dimension of the display in our numerous examples.

What is especially striking is that the Hegelian analysis of <u>any</u> topos turns out to involve <u>graphic</u> monoids which are in fact bicategories. Thus, the organization of any branch of knowledge, insofar as it can be mathematical (i.e. teachable), may in some measure reflect itself in graphic displays. Though proposed [0] nearly 200 years ago, the Hegelian method of analysis has been

widely under-utilized since then; "conflicting" ideological claims either that it is inconsistent or that it is too wonderfully fluid to be made mathematical have conspired to prevent its being widely taught. We believe that we have through modest examples shown it to be consistent (and non-trivial) and that much of the method should be made mathematical, which would help those who seriously want to use it, even that part which remains fluid.

By a constant c in a graphic monoid is meant an element such that $cx = c$ for all x. The three element monoid with two constants ∂_0, ∂_1 (so $\partial_i \partial_j = \partial_i$) has as its applications all the reflexive directed graphs; that example plays a central role in [1,2] and suggested the name. Toposes of applications of such "constant" graphics with more than two constants were investigated in [2], partly as a vehicle for explaining some basic topos theory and partly to determine how they were different from the two-constant cases in which $x\partial_0, x\partial_1$ denote the beginning and ending points of an arbitrary directed edge x. In the course of that work, the identity $xyx = xy$ was discovered as the least common generalization of constant ($x = c$) and identity ($x = 1$); later I learned that it had been briefly mentioned as a purely formal generalization in [3], where the finiteness was noted, and that in [4] a partial structure theorem for such monoids was proved as well as a structure theorem for certain more general monoids using these as one of the ingredients. (As for finiteness, it is immediate that the free graphic monoid on a finite set of letters consists of all words without repetitions, of which there are only $n! \sum_{i=0}^{n} \frac{1}{i!}$). So far I have not found any previous discussion of applications (in either sense).

In this paragraph (and the next) we make some imprecise remarks about possible uses. Retrieving stored knowledge presupposes some consciousness of the structure it has; this structure is in its particularity fixed by the storage process itself (and

in its generality is partly a reflection of the content, i.e. of the nature of the knowledge stored). Thus in both retrieval and storage one needs to be explicitly aware of the kind of structure involved. Here we are momentarily accenting the "passive" aspect of the structure, the kind of structure that both codomain and domain of more "active" operations such as rewrite must have ("peeking" may be definable). Now it is commonly recognized that commutative operations such as Boolean intersection are involved, but also "something further". We here speculate that non-commuting systems of idempotent operations may capture some of the further subtlety. The arrangement of shelves in any science library shows that topological algebra ≠ algebraic topology and chemical physics ≠ physical chemistry, although these are in some sense "intersections". A feature which seems to be present is that a sub-branch b is not only a subset but reflects things x (not necessarily in b) to a part bx of b which is most relevant to x (bx is a single element in the generic case of G(-,I) but the idea retains force in general applications).

As another example, we could assign to every page of every book the title page of the book that it is in; clearly this operation specifies the set of all title pages, but much more. Such idempotent operations need not commute but on the other hand would have a rather strong commutation relation reflecting the hierarchical structure of empty documents within folders within disks.... We have pursued the investigations summarized here in the hope that the "graphical" identity may capture many instances of this commutation relation. This hope was strengthened by the recent discovery that that identity arises in the Hegelian scheme of knowledge. It is said that the German philosopher Hegel, building on the work of Aristotle and in opposition to the eclectic listing of categories of sciences by his "metaphysical" predecessor Wolfe, proposed to generate the main categories by a single dialectical process. The great mathematician Grassmann, partly inspired by Leiniz, also emphasized the dialectical method in

building up his geometrical theory of extensive quantities. What striking contrast between these, who advanced both knowledge and its organization, and those to whom $x \in x$ is a big issue and who lead us astray with library-catalogue paradoxes, when more conscious access to libraries is what is needed! [2)]

II. Elementary Consequences of the Basic Identity, with special reference to ideals

We begin our calculations by pointing out some remarkable consequences of the graphic identity

$$aba = ab.$$

For any right action X of any monoid M, there is for any element x the <u>stabilizer</u>

$$Stab(x) = \{a \in M \mid xa = x\}$$

<u>PROPOSITION 1</u> If M is a graphic monoid, then the stabilizer of any element x of any application X is a <u>saturated</u> submonoid: $ab \in Stab(x) \Longrightarrow a,b \in Stab(x)$.

Proof: $xab = x \Longrightarrow xa = xaba = xab = x$ and $xb = xabb = xab = x$.

For any action the part fixed by all M is a (trivial) subaction, but the part fixed by a single $a \in M$, which for idempotent a satisfies

$$Xa = \{x \in X \mid xa = a\} ,$$

is usually only a sub<u>set</u> (it <u>is</u> a functor of X).

<u>PROPOSITION 2</u> If M is graphic and $a \in M$ and if X is any application of M, then Xa is actually a sub-application, i.e. $x \in Xa \Longrightarrow xb \in Xa$ for all $b \in M$.

Proof: $xa = x \Longrightarrow (xb)a = xaba = xab = xb \Longrightarrow xb \in Xa$

One of the most powerful consequences of the graphic identity is that

every left ideal is a right ideal

which follows from the next proposition, using the fact that every ideal of either kind is a union of principal ideals.

PROPOSITION 3 For any element of any graphic monoid M

$$aM \subseteq Ma$$

Proof: For every x there is an element x^a for which

$$ax = x^a a,$$

namely, we can take $x^a = ax$.

Since every element of a graphic monoid is idempotent, it follows trivially that

every left ideal S is idempotent

in the sense that $SS = S$. For a general monoid, this would be equivalent to "for every a, there are u,v for which $a = uava$". This would include all groups, and also the monoid of all endomaps of a 2-element set, which figures in [2]. Perhaps much of what follows could be generalized to all monoids satisfying the two boxed axioms above, but if we assume idempotence of elements, it can be shown that $aM \subseteq Ma$ implies the graphic identity.

Often Ma is much bigger than aM, but as a right ideal it is a finite union $\bigcup b_i M$ of principal right ideals. The smallest number $\#(a)$ of b_i required could be considered as a crude measure of the size of a.

PROPOSITION 4 $Ma = \bigcup b_i M$ iff

1) $b_i = b_i a$ for all i

2) for all x, $xa = b_i x$ for some i .

In particular, one of the b_i must be a itself.

Proof: $xa = b_i y$ for some y so $xa = b_i xa$ by idempotence. Thus $xa = b_i axa$ by 1) so $xa = b_i ax = b_i x$. Taking $x = 1$ proves the last remark.

Normally a principal ideal can have more than one generator, but in a graphic the elements are faithfully represented by right ideals:

PROPOSITION 5 In a graphic monoid, $aM = bM \Longrightarrow a = b$.

Proof: We have $a=bx$ and $b = ay$, hence by idempotence $a = ba$ and $b = ab$. But $a = ba = bab = bb = b$.

For principal left ideals we do not have faithfulness but we do have, since $Ma = Mb$ iff $a = ab$:

PROPOSITION 6 In a graphic monoid, $Ma = Mb$ iff $a = ab$ and $b = ba$ iff $Stab(a) = Stab(b)$ iff a,b are the images of ∂_0, ∂_1 under a homomorphism from the three element monoid with 2 constants.

Note that $aM \cap bM$, while a right ideal, is not usually a principal right ideal and is often even empty. But for principal left ideal this situation is simpler:

PROPOSITION 7
$$Mab = Ma \cap Mb$$
$$M1 = M$$

for any graphic monoid. Hence $Mab = Mba$.

Proof: $Mab \subseteq Mb$ is clear. By the graphic identity, we also have $Mab \subseteq Ma$. If an element x is in both Ma and Mb , then $x = xa$ and $x = xb$ by idempotence, so $x = xb = xab \in Mab$.

As Kimura [4] proved and used, the image CM of the homomorphism $M \longrightarrow$ (left ideals, $\cap$) thus defined is actually the

universal homomorphism to any commutative graphic monoid (=semilattice). Schanuel (unpublished) showed, as suggested by Propositions 1 and 6, that this semi-lattice reflection CM can alternatively be constructed as part of the set of all saturated submonoids under the join operation on such (note that $Ma \subseteq Mb$ iff $Stab(a) \supseteq Stab(b)$).

Now we recall that in the topos of all applications of M, the truth-value application Ω is the one consisting of all right ideals of M, under the action of each $b \in M$ defined at A by

$$A{:}b = \{x \in M \mid bx \in A\}$$

which is easily seen to be another right ideal if A was. The universal use of Ω is: if $Y \subset X$ is any sub-application, then $X \xrightarrow{\varphi} \Omega$ defined by

$$\varphi x = \{a \in M \mid xa \in Y\}$$

is an M-equivariant morphism of applications, and the unique one for which

$$x \in Y \iff \varphi x = \text{true}$$

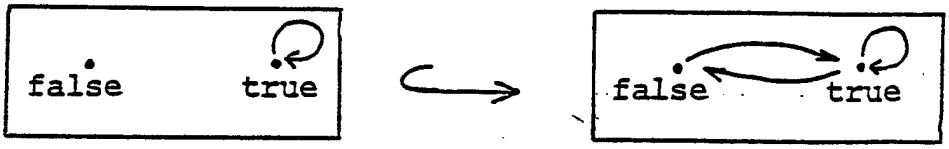
(where true = $M \in \Omega$) holds for all x in X. In general φx is thought of as the truth-value of the statement "$x \in Y$", which value just consists of all available acts which bring about actual truth. For example, in the case where applications = directed graphs, there are five truth-values, two of which are points, one is a loop at true, and the other two are edges connecting (in the two directions) true with false = $\emptyset$.

In the case of a graphic monoid we have shown (Proposition 3) that every left ideal is a right ideal. Even more remarkably, if we consider the sublattice $\Omega_{left} \subset \Omega$ (of the distributive lattice of all right ideals) which consists of the left ideals, we have

PROPOSITION 8 For a graphic monoid M, $\Omega_\ell \subset \Omega$ is a sub-application.

Proof: If S is a left ideal and $a \in M$, then $S:a = \{b \mid ab \in S\}$. We must show that this is again a left ideal. So suppose $ab \in S$ and that $c \in M$; we must show $cb \in S:a$, that is that $acb \in S$. But $acb = (aca)b = acab \subseteq Mab \subseteq S$ since S itself was a left ideal.

Even though the inclusion of posets $\Omega_\ell \subset \Omega$ has both a left adjoint $(A \longmapsto MA)$ and a right adjoint, neither of the latter is a morphism of applications. For example, for directed graphs (where $\circlearrowright <$ true in the ordering, which we suppress) the inclusion in question is



which admits no graph-theoretic retraction (order-preserving or not). Note that $aM \subseteq bM \Rightarrow Ma \subseteq Mb$.

Although applications in general do not have left actions, we can ask: For which inclusions $Y \subset X$ of applications does the corresponding characteristic map $\varphi : X \longrightarrow \Omega$ actually factor through the sublattice $\Omega_\ell \subset \Omega$ of left ideals? In the example of directed graphs, the above picture shows the answer to be: those subgraphs Y of the graph X for which no directed edge of X enters Y or leaves Y except on excursion, i.e. $x\partial_0 \in Y \Leftrightarrow x\partial_1 \in Y$ for all x.

Now in the generic application $X = M$, the left multiplication by a may be considered as the reflection of an arbitrary x to (the "most relevant element of"?) the fixed point set Xa In a particular application X, left multiplication by a is usually not defined. However, by proposition 2, $Xa \subset X$ is a

sub-application, and hence by the universal property of Ω there is a unique characteristic map $\varphi_a : X \longrightarrow \Omega$, and we have $aM \subseteq \varphi_a x$ for all x, for even $Ma \subseteq \varphi_a x$. We may ask, when is $\varphi_a(x) \in \Omega_\ell$? By definition

PROPOSITION 9 $\varphi_a(x) \in \Omega_\ell$ iff

$$\forall b, \lambda \in M \left[xba = xb \Longrightarrow x\lambda ba = x\lambda b \right]$$

PROPOSITION 10 If $X = M$ and if M consists only of constants and 1, then $\varphi_a x \in \Omega_\ell$ for all x, $a \in M$.

Throughout this paper we consider only the category of right actions or "applications" (categories of left actions are treated very briefly in the examples in [2] and have rather different properties). Thus it must constantly be kept in mind that whenever we attribute a property such as "connectedness" to a left ideal S, we are using our proposition 3 to consider S as an object in the category of (right) applications-connectedness of S as a left action would mean something quite different! Similarly, when the set Ω_ℓ of left ideals is considered as an object in a category, it will be (either as a lattice or) according to proposition 8 as an application.

III. Elementary Examples and their Intuitive Displays

In preparation for listing some examples of graphics, let us make explicit some facts about the role of constants.

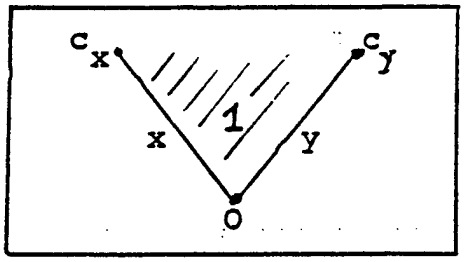
PROPOSITION 11 Every graphic monoid contains constants.

Proof: Since we have assumed finiteness, let c be the product, in some chosen order, of all the elements of the monoid. Then $cx = c$ for any x, since x already occurs first as a factor of c, and the basic identity cancels second occurrences.

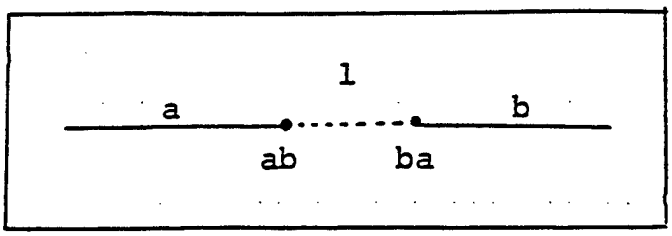
For example, the free graphic monoid on n generators has $n!$ constants, since all words of maximal length are distinct. On the other hand, all those can be collapsed to one without imposing any further relations between words of shorter length. Thus (not only commutative) examples may have a unique constant.

PROPOSITION 12 If c is a constant, then so is ac for any a. Thus Ma includes all constants, hence any non-empty left ideal contains all constants. Also if there is a unique constant o, we have $ao = o$ for all a.

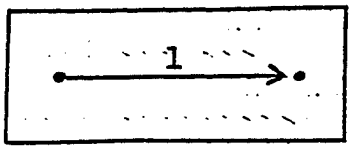
The left action of M on the set T_o of all constants of M may thus fail to be faithful. However, we can always adjoin new constants, for example via the sub-representation $M \cup X$ of the faithful left regular representation of M on $X = M$. If we do that to the four-element free semilattice on two generators x,y, we get a six-element graphic whose display will turn out to be the two-dimensional picture



Of course any free graphic monoid does act faithfully (on the left) on its constants. For example the five-element free graphic monoid on two generators a,b has the two constants ab and ba, on which the generators act by interchanging them; however, its display will turn out to be the one-dimensional:



The graphic monoid $\mathbb{A}_1$ with only three elements, two of which are constant, is displayed

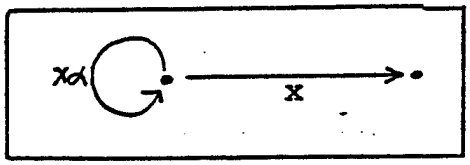


and all its applications are "one-dimensional", being directed graphs. It is of wide use in analyzing more complicated graphics, for example, consider the graphic monoid M which is freely generated by two elements α , ∂_1 subject to the one relation $\partial_1 \alpha = \partial_1$ and define $\partial_0 = \alpha\partial_1$. Then

$$\partial_0\partial_1 = \alpha\partial_1\partial_1 = \partial_0$$

$$\partial_1\partial_0 = \partial_1\alpha\,\partial_1 = \partial_1\alpha = \partial_1$$

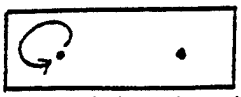
so that any M-application has in particular an underlying directed graph, but is more in that α also acts on the directed edges. In addition to the defining relation, we have $\partial_0\alpha = \alpha\partial_1\alpha = \alpha\partial_1 = \partial_0$ so that both ∂_i remain constants even in M . The definition of ∂_0 says that any $x\alpha$ ends at the beginning of x , but moreover $\alpha\partial_0 = \alpha^2\partial_1 = \partial_0$ so that $x\alpha$ is a <u>loop</u> at $x\partial_0$. Thus every edge x in an application carries with it a picture

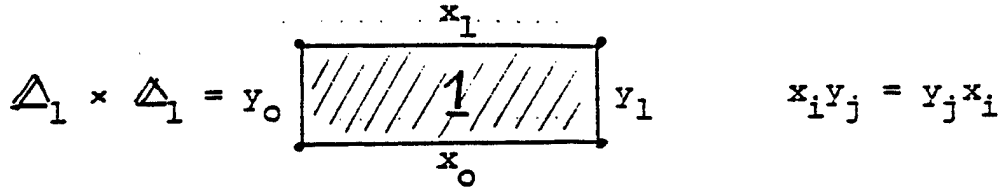


;

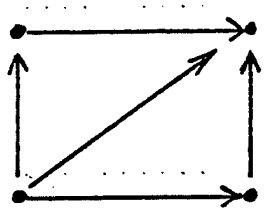
if x is interpreted as a process, we might consider $x\alpha$ as the "preparation" necessary for x . In order to represent M faithfully by endomaps, consider one more constant * together with $\partial_0\partial_1$ and define an operation on this three-element set by $\alpha(\partial_0) = \alpha(\partial_1) = \partial_0$, $\alpha(*) = *$. The left-ideal lattice Ω_ℓ has four elements

$$\emptyset \subset M\partial_0 = M\partial_1 \subset M\alpha \subset M$$

but $M\mathcal{A} =$  is not "connected", which will mean that even as a graphic in its own right, M must be displayed as one-dimensional. This contrasts with

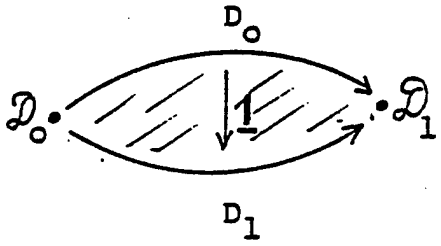


a two-dimensional, nine element graphical monoid, which like the above M also receives a homomorphism $\Delta_1 \longrightarrow \Delta_1 \times \Delta_1$, say the diagonal. Along the latter, we also get an underlying graph, whose display is



In general, if every homomorphism $\Delta_1 \longrightarrow M$ is assigned a color, then all the underlying graph structures of M could be simultaneously displayed.

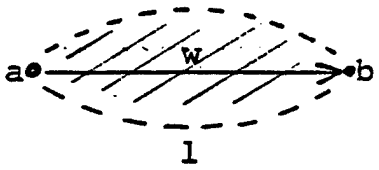
For another important example, recall that graphs underlie the theory of categories, but that there are also 2-categories; underlying the latter are <u>2-graphs</u>, the generic example of which is



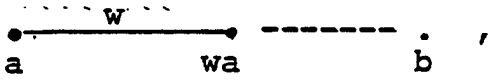
This can be made into a five element (four generator) graphical monoid by defining $\mathcal{D}_i \mathcal{D}_j = \mathcal{D}_i$, $D_i D_j = D_i$, $D_i \mathcal{D}_j = \mathcal{D}_j$, $\mathcal{D}_i D_j = \mathcal{D}_i$ Every 2-category (for example the 2-category of all graphics, all functors between these, and all natural transformations between those) has an underlying application of this monoid, in which

the $\mathcal{D}_{D_i}$ are the domain and codomain "functors" of any "natural transformation" $\mathcal{D} = \mathcal{D} \cdot 1$ and FD_i are the domain and codomain "categories" of any "functor" F. The lattice Ω_ℓ turns out to be a linearly-ordered set isomorphic to $\{-\infty < 0 < 1 < 2\}$ where 0 stands for the constant ∂_i but 1 stands for the left ideal $\bullet \longrightarrow \bullet$ which is already connected as a right ideal, hence (by the general theory to be described presently) the graphic itself has a two-dimensional display.

If to a nontrivial graphic monoid we adjoin a new identity element, so that the original monoid becomes a connected left ideal in the new monoid, we get again a graphic monoid of dimension at least two. If we do this to Δ_1, and denote the original identity element by w, we see



that w is more of a "core" than a "boundary", and moreover that, since this is a homomorphic image of



dimension can be increased by homomorphic image. Since $w\partial_i = \partial_i$, in the underlying-graph display of M the cloud 1 condenses into another arrow parallel to w.

In order to describe a certain class of examples, two more propositions will be helpful.

PROPOSITION 13 The lattice Ω_ℓ of left (=bi) ideals in a graphic monoid M is linearly ordered iff for every pair a,b of elements in M

$$a = ab \quad \text{or} \quad b = ba \ .$$

Proof: This is the condition that $Ma \subseteq Mb$ or $Mb \subseteq Ma$, i.e. that the (semilattice) commutative reflection CM be linearly ordered. But the left ideals of CM are included surjectively into the left ideals of M, and the left ideals of a linear semilattice are clearly linearly ordered.

PROPOSITION 14 (Schanuel) Suppose that the endomorphism monoid of an object A in a category (such as $\overline{M}$) satisfies the graphic identity, and that B is any other object. Then there is at most one splittable epimorphism $A \xrightarrow{p} B$. In case A,B are retracts of a common graphical object I with idempotents a,b then p exists iff $Mb \subseteq Ma$, where M is the endomorphism monoid of I.

Proof: Suppose p has splitting section s, but that also q has splitting section i; that is $ps = 1_B = qi$. Then of course sp and iq are idempotents at A, but since A is graphic also ip and sq are idempotents. Better

$$sq = s(pi)q = (sp)(iq)(sp) = s(piqs)p = sp$$

so that $q = p$ because s is a monomorphism. It is easily checked that at least one p exists iff $b = ba$, in the $\overline{M}$ case.

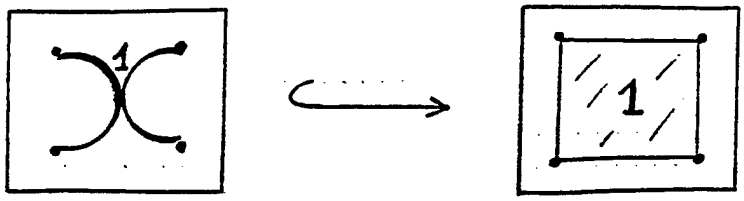
Thus in any graphic the subcategory of all splittable epimorphisms forms a poset. If

$$A = B_n \xrightarrow[p_n]{} B_{n-1} \longrightarrow B_{n-2} \longrightarrow \cdots \longrightarrow B_o \longrightarrow B_{-\infty}$$

is any linear family of splittable epimorphisms in any category, and if we consider for each k any non-empty finite set of sections $B_{k-1} \xhookrightarrow[s]{} B_k$ for p_k, then the submonoid of endomorphisms of A obtained by considering all composites will be a graphical monoid. Special interest will attach in part IV. to the case where we consider two sections for each p_k.

Note that the unique retraction $I \longrightarrow aM$ "represents" on the level of elements all the unique inclusions $X_a \hookrightarrow X$ in the topos of applications of M.

The (one-dimensional) graphic monoid with four constants and five elements (which was described as a "bare unity" in [2]) can be embedded in the two-dimensional $\Delta_1 \times \Delta_1$; the one dimensional connection might be displayed as



Another interesting embedding is

PROPOSITION 15 The free graphic monoid F on two generators a,b can be embedded in $\Delta_1 \times \Delta_1 \times \Delta_1$.

Proof: Note that $M = \Delta_1 \times \Delta_1$ has a pair of elements f,s such that $s \neq fs = sf \neq f$. For any such M , F can be embedded in $M \times \Delta_1$ by sending $a = \langle f, \partial_0 \rangle$, $b = \langle s, \partial_1 \rangle$.

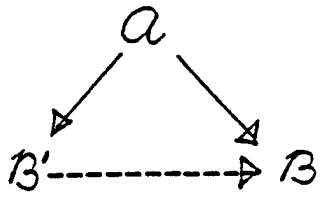
IV. Unity and Identity of Opposites in Bicategories and precise Definition of the refined and coarse Dimensions of Displays

In order to clarify the notion of dimension which arose in our intuitive displays of graphics, as well as to provide an infinite number of examples of graphics arising from non-idempotent mathematical structures, consider the following

DEFINITION A functor $\mathcal{A} \longrightarrow \mathcal{B}$ will be called a unity-and-identity-of-opposites (UIO) iff it has both left and right adjoints and one of the latter is full and faithful (hence both are). Then, denoting by L and R the two idempotent endofunctors of $\mathcal{A}$ obtained by composition, we have also $L \dashv R$ and $LR = L$, $RL = R$.

The two adjoints are the inclusions of two opposite sub-categories united in $\mathcal{A}$, yet identical with $\mathcal{B}$. The terminal functor $\mathcal{A} \longrightarrow \mathbb{1}$ is a UIO iff $\mathcal{A}$ has both initial and terminal objects; the latter may be called non-being and pure being resp., and in general L is "non" whatever attribute (of $\mathcal{A}$) R is the "pure" form of. If $\mathcal{A}$ is a topos then $\mathcal{B}$ will automatically be a topos as well; this applies to our fundamental class of examples,

where $\mathcal{A}$ is the category of all applications of a given graphic. In case $\mathcal{A}$ is a topos, R is called the $\mathcal{B}$-sheafification, and "non" sheaves may be called $\mathcal{B}$-skeletal. The set of all UIO's with a given $\mathcal{A}$ forms a poset with respect to the "greater than" ordering



This poset is often small even when $\mathcal{A}$ is large and is often a complete lattice, as is shown in a forthcoming joint paper with Kelly [5] . For example

PROPOSITION 16 If $\mathcal{A}$ is a category of all right actions (on sets) of a small category C , then the poset of UIO's with domain is equivalent to the poset of all idempotent two-sided ideals in the category C , with the empty ideal corresponding to $\mathcal{A} \longrightarrow \mathbb{1}$.

Corollary: For the category $\mathcal{A}$ of all applications of a given graphic monoid M the poset of all UIO's is parameterized by the poset of all left ideals of M . In more detail, if S is a left ideal of M , then an application X is an S-sheaf iff every morphism $S \longrightarrow X$ in $\mathcal{A}$ is of the form $s \longmapsto x \cdot s$ for a unique element x of X , and on the other hand the S-skeleton $L_S(X) \subset X$ of any application X is given by

$$L_S X = \bigcup_{s \in S} Xs$$

i.e. all those elements of X that are fixed by some $s \in S$. Moreover, (since idempotence is automatic and quite unlike the general case) (not only the suprema but also) the infima in this finite (distributive!) lattice are computed as ordinary (unions and) intersections.

We will attribute refined dimension $\leq S$ to all applications X which satisfy the "negative determination" $L_S X \xrightarrow{\sim} X$.

In particular, $\emptyset$ will also be called of dimension-∞ and T_0 = the set of all constants of M determines the subtopos $\mathcal{B}_0$ of all "codiscrete" applications so that 0-dimensional means "discrete": We will assume that M has at least two constants, which implies that Ω is connected ($\pi_0\Omega = 1$) and that the "components" functor $\mathcal{A} \xrightarrow{\pi_0} \mathcal{B}_0$ (extra left adjoint to the discrete inclusion) preserves finite products [1,2] . To define coarse dimensions 1,2,... we will use the following

DEFINITION: If $S \subseteq T$ are left ideals, say that T resolves the opposites of S , in symbols

$$S \overset{a}{<<} T$$

iff every S-skeletal application is a T-sheaf, i.e. iff $R_T L_S = L_S$ Because of the nice properties of intersection mentioned in the corollary to Proposition 16, there is for every S a smallest S' which resolves the opposites of S ; we may call S' the "Aufhebung" of S . Then the Aufhebung of pure being versus non-being is pure becoming versus non-becoming, i.e. codiscrete (chaotic) versus discrete, since if $\emptyset$ is to be a T-sheaf, then there can be no maps $T \longrightarrow \emptyset$, i.e. T must be non-empty, but by Proposition 12, T_0 (= the set of all constants of M) is the smallest non-empty left ideal; thus $(-\infty)' = 0$ as claimed. Since, intuitively, one-dimensional figures are the dimensionally-smallest ones which permit connecting all those points that can be connected, still more satisfying is

PROPOSITION 17 $0' = 1$. That is, $\pi_0 L_T = \pi_0$ iff $R_T L_0 = L_0$. Thus T_1 is characterized as the smallest left ideal of M which is connected as a (right) application of M .

Proof: Composite adjoints are adjoint composites. Or, if discrete applications D are to be T-sheaves, then every $T \longrightarrow D$ must come from an element of D ; but elements of D are constant (non-becoming), hence every $T \longrightarrow D$ must be constant (e.g. for $D = 2$), hence T must be connected.

Corollary: If $M \setminus \{1\}$ is not connected, then M is one-dimensional, whereas if $M \setminus \{1\}$ is connected and is the "Aufhebung" of some S which is in turn an Aufhebung..., then M is at least two-dimensional.

Here the dimension of M itself is defined in terms of the length of the sequence $T_{n+1} = T'_n$; experience [6] with other examples suggests that this length is the dimension for small dimensions and a simple function of it for higher dimensions.

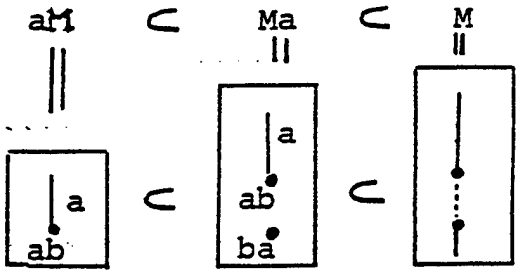
PROPOSITION 18 If M is the free graphic monoid on $k \geq 2$ generators, then dim $M = 1$.

Proof: Since "first letter of a word" is well-defined,

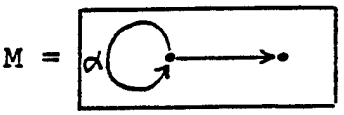
$$M \setminus \{1\} = \sum_{i=1}^{k} a_i M$$

is a disjoint sum in the category of applications, hence not connected.

While principal right ideals are connected, principal left ideals need not be, for example, Ma in the free example on a,b :



An even smaller example of an "infinitesimal dimension" is provided by



where  is a left ideal. But note that a left ideal which contains a connected left ideal is itself connected, for any t can be moved to a constant by the right action of a constant.

Now consider any category $\mathcal{A}$ with initial and terminal objects $\emptyset, 1$ and a double resolution of the latter by $\mathcal{C}, \mathcal{B}$ which successively climb $\mathcal{A}$-ward $\mathcal{A} \dashrightarrow \mathcal{B} \dashrightarrow \mathcal{C} \dashrightarrow 1$. Let r = pure $\mathcal{C}$, ℓ = non $\mathcal{C}$, R = pure $\mathcal{B}$, L = non $\mathcal{B}$. The first resolution means $r\emptyset = \emptyset$ (which implies $\mathcal{A}_{77} \subset \mathcal{C}$ if $\mathcal{A}$ is a topos) while the second, $R\ell = \ell$ means that there are three (rather than four) subcategories of $\mathcal{A}$ "identical" with $\mathcal{C}$. Assume for simplicity that also $\ell 1 = 1$. Consider the category $\mathcal{M}$ of all endofunctors of $\mathcal{A}$ definable by composition from these and all natural transformations definable from the adjunction morphisms. $\mathcal{M}$ is a finite non-symmetric monoidal category, and there is only one object $q = Lr$ in $\mathcal{M}$ which does not have either a left or a right adjoint in $\mathcal{M}$ - it comes from the third embedding of $\mathcal{C}$ in $\mathcal{A}$.

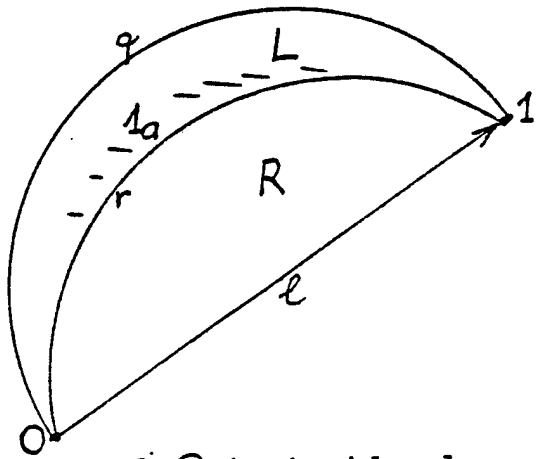
PROPOSITION 19 The objects of $\mathcal{M}$ under composition constitute (up to equivalence) a graphic monoid of ($\leq$) eight elements which has five left ideals

$$\emptyset \subset \{0,1\} \subset [\ell, q, r] \subset [L, R] \subset [1_{\mathcal{A}}]$$

(where we have shown only the elements new at each stage). The middle of these (generated by any lower case letter) is already connected (by the right action of 0 !).

Thus the display of $\mathcal{M}$ is apparently

Hegelian "taco", a display of the 3-dimensional 8-element graphic monoid $\mathcal{M}$



which reminded some of a taco: All the meat of $\mathcal{A}$ is inside $1_{\mathcal{A}}$, while there are two identical faces L,R with a common edge ℓ and separate (but identical) edges q,r.

To finish the proof that $\mathcal{M}$ is really three-dimensional, we need only show that the Aufhebung of $[\ell,q,r]$ is just $[L,R]$, i.e. does not somehow jump all the way to the top $[1_a]$ of the dimension lattice as happens in other examples. But $S = [\ell,q,r]$ is actually <u>principal</u> $S = \mathcal{M}\ell$, while for such principal ideals it is easily seen that $L_S X = X\ell$ for all applications X of $\mathcal{M}$; thus for X to be S-skeletal merely means that all elements of X are fixed by the right action of ℓ. Suppose X is all fixed by ℓ; we must show that X is already an $[L,R]$-sheaf, so consider any morphism $[L,R] \xrightarrow{f} X$ of applications, which we must show comes from a unique complete element of X. The uniqueness is immediate, since if x,y are any two elements of X with the same $[L,R]$ part f, we have $xt = yt$ for all $t \in [L,R]$, but $t = \ell$ is such and we have already assumed X fixed by ℓ: thus $x = x\ell = y\ell = y$. For the existence of an x extending the partial element f, note that, while a general application X consists of a complicated interlocking system of "tacos", the skeletal condition means that these are all degenerated with $x = x\ell = xq = xr$, i.e. all three "edges" of any element x coincide; this implies also $xL = x\ell L = x\ell = x$ and similarly $xR = x$, leaving only the endpoint operators $x0$, $x\cdot 1$ acting possibly non-trivially: to sum up, such a skeletal $\mathcal{M}$-application is in essence just a directed graph. Now a partial element f defined only on the faces $[L,R] = \mathcal{M}L$ has in particular all its values fixed by ℓ due to the skeletal condition, so

$$f(L) = f(L)\ell = f(L\ell) = f(\ell)$$

$$f(R) = f(R)\ell = f(R\ell) = f(\ell)$$

the last being true because of the Aufhebung condition $R\ell = \ell$ in the definition of $\mathcal{M}$ itself. Thus the element $x = f(\ell)$ seems the likely candidate for a complete (degenerately) three-dimensional element whose restriction to the seven-element ideal $[L,R]$ could be f itself. Thus we try to show

$$\boxed{f(\ell)a = f(a)}$$

for all seven $a \in [L,R]$. For $a = L,R$ we have by the above

$$f(\ell)L = f(L)L = f(L)$$

$$f(\ell)R = f(R)R = f(R).$$

(both of course equal to $f(\ell)$). For the two constants $a = 0,1$ we have $f(\ell)a = f(\ell a) = f(a)$ since $\ell 0 = 0, \ell 1 = 1$. For the remaining three $a = \ell,q,r$ the case $a = \ell$ is tautologous, and for $a = r,q$ we have

$$f(\ell)r = f(\ell r) = f(\ell)$$

$$f(\ell)q = f(\ell)Lr = f(\ell Lr) = f(\ell)$$

so that we are reduced to showing that

$$f(r) = f(\ell) = f(q).$$

For this we need to use that f is defined also on the two-dimensional L,R since otherwise these could be three different edges (with the same endpoints $f(0),f(1)$) of the directed graph. But since $f(R) = f(\ell)$,

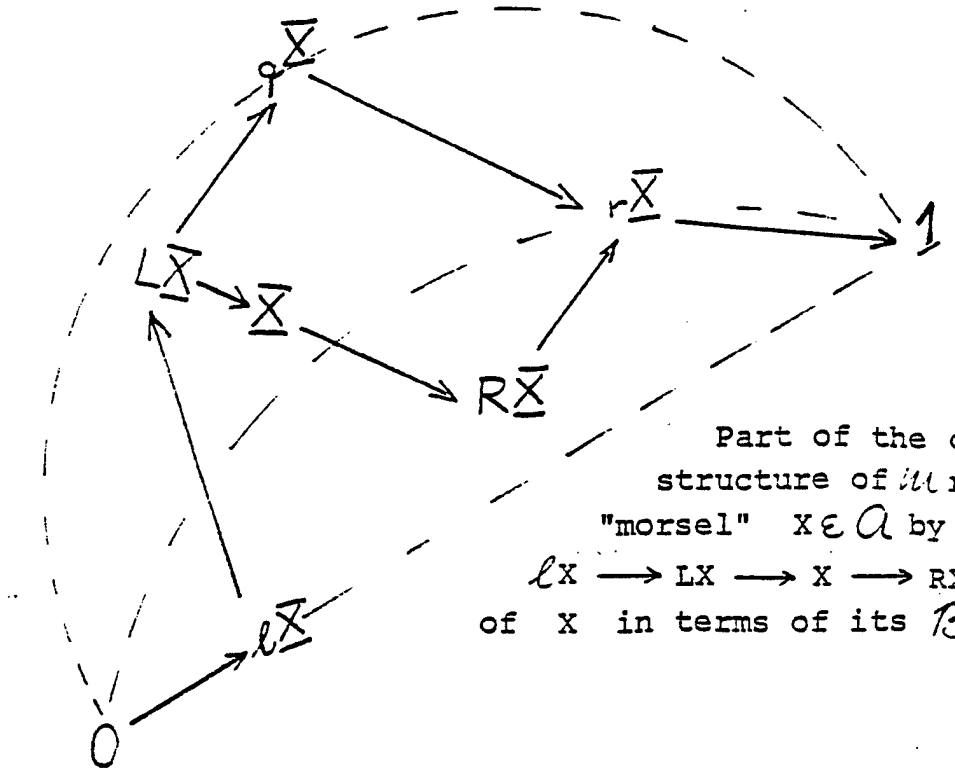
$$f(r) = f(Rr) = f(R)r = f(\ell)r = f(\ell r) = f(\ell)$$

and since $f(L) = f(\ell)$,

$$f(q) = f(Lr) = f(L)r = f(\ell)r = f(\ell r) = f(\ell)$$

so the proof is done.

Of course the above display does not show that $\mathcal{M}$ is a monoidal <u>category</u>, not just a graphic monoid; if $X \in \mathcal{A}$ is any "morsel", then the horizontal slice through the "taco" at X actually has canonical morphisms of $\mathcal{A}$ (indexed by $\mathcal{M}$), which are roughly the "Moore-Postnikov" analysis of X in case $\mathcal{A}$=combinatorial topology, as follows:



Part of the category structure of $\mathcal{M}$ revealed at the "morsel" $X \in \mathcal{A}$ by the analysis

$\ell X \longrightarrow LX \longrightarrow X \longrightarrow RX \longrightarrow rX$

of X in terms of its $\mathcal{B}$ and $\mathcal{C}$ reflections.

measuring how closely the various reflections of X (into the grasped stages $\mathcal{C}$, $\mathcal{B}$) succeed in approximating it.

PROPOSITION 20 The "slice" obtained by omitting 0,1 from $\mathcal{M}$ is as a graphic monoid isomorphic to a six-element submonoid of the monoid of all order-preserving endomaps of a three-element linearly-ordered set; namely omitting 001, 002, 112, 122 from the latter corresponds to the former via $0 \longmapsto \ell$, $1 \longmapsto q$, $2 \longmapsto r$. (Note that ℓ,q,r have become constants through this omission).

The proof is left to the interested reader.

NOTES

1) This research was not supported by any granting agency.

2) I am not a "Hegelian", since I reject Hegel's Objective Idealism. But Hegel's partly-achieved goal of developing Objective Logic (as a component of the laws of thought at least as important as the Subjective Logic commonly considered to be "all" of Logic) is in a way the program which the whole body of category theory has been carrying out within mathematics for the past 50 years. It was because of some discoveries in the foundations of homotopy theory that I began a few years ago the study of The Science of Logic, attempting to extract the "rational kernel" which, insofar as it truly reflects laws of thought, should be useful to us in investigations like the one summarized in this paper.

REFERENCES

[o] Hegel, G.W.F., The Science of Logic, 1812-1813-1816, another shorter version in his Encyclopedia, Heidelberg 1817, both now in several translations.

[1] Lawvere, F.W., Categories of spaces may not be generalized spaces, as exemplified by directed graphs, Revista Colombiana de Matemáticas, 20 (1986), 179-188.

[2] Lawvere, F.W., Qualitative distinctions between some toposes of generalized graphs, to appear in Contemporary Mathematics 92 (1989) (Proceedings of AMS Boulder Conference on Category Theory and Computer Science 1987).

[3] Schützenberger, M.P., Sur certains treillis gauches, C.R. Acad. Sc. Paris 224 (1947), 776-778.

[4] Kimura, N., The structure of idempotent semigroups I, Pacific J. Math. 8 (1958), 257-275.

[5] Kelly, G.M., and Lawvere, F.W., On the complete lattice of essential localizations, to appear in the Proceedings of the Louvain meeting in honor of René Lavendhomme's 60th birthday. (1989).

[6] Zaks, M., Doctoral Thesis on new aspects of simplicial sets, in preparation at Macquarie University, North Ryde, N.S.W. (Australia).

The stratified loose semantics: An attempt to provide an adequate algebraic model of modularity

Michel Bidoit[1]

Laboratoire de Recherche en Informatique
C.N.R.S. U.A. 410 "Al Khowarizmi"
Université Paris-Sud – Bât. 490
F – 91405 ORSAY Cedex France
e-mail: mb%FRLRI61.bitnet@cunyvm.cuny.edu

Abstract

One of the most obvious applications of algebraic methods to software technology are algebraic specifications. In this paper we investigate how far the development and the reuse of modular software can effectively be supported by algebraic specifications. We show that modularity cannot be modelled as easily as one may expect, and we introduce a new semantic framework, *the stratified loose semantics*, which can be considered as a generalization of both initial and loose semantics and which is used to define the formal semantics of the **Pluss** algebraic specification language.

1 Introduction

The problem considered in this paper concerns the algebraic specification of reusable, modular software. Since the pioneer work of [11], algebraic specifications have been advocated as being one of the most promising approach to enhance software quality and reliability. Algebraic specifications proved to be useful not only to formally describe complex software systems, but also to prototype them (e.g. by transforming axioms into an equivalent set of rewriting rules), and to prove the correctness of these software systems (w.r.t. their formal, algebraic specification). More recently, it has also been shown that algebraic specifications provide suitable means to compute adequate test sets for the described software systems, and that they provide also a formal basis to promote software reusability (to decide whether or not some software is reusable for some specific purposes being shown equivalent to the "comparison" of the formal specification of the software to be reused with the formal specification of the software to be written). An important aim of the research activity in the area of algebraic specifications is to provide adequate concepts, languages and tools to cover the whole software development process and to establish their

[1]This work is partially supported by ESPRIT Project 432 METEOR and C.N.R.S. GRECO de Programmation.

mathematical foundations.

In this paper we shall focus on the links that can (should) be established between a structured specification and the corresponding software implemented using a modular programming language such as Ada, Clu or ML. The problem considered is to define an algebraic semantic framework such that the various pieces of the specification can be related to the various modules of the implementation and such that the global correctness of the implementation can be established from the local correctness of each software module w.r.t. its specification module.

2 Modularity and loose algebraic specifications

To better understand why and how far both the modularity of the specification and the modularity of the software interact together as well as the need for a new approach to the semantics of algebraic specifications, we shall first briefly recall the main underlying paradigm of the loose approach.

A specification is supposed to describe a future or existing system in such a way that the properties of the system (**what** the system does) are expressed, and the implementation details (**how** it is done) are omitted. Thus a specification language aims at describing classes of correct (w.r.t. the intended purposes) implementations (realizations). In contrast a programming language aims at describing specific implementations (realizations). In a loose framework, the semantics of some specification $\tilde{S}$ is a class $\mathcal{M}$ of (non-isomorphic) algebras. Given some implementation (program) P, its correctness w.r.t. the specification $\tilde{S}$ can then be established by relating the program P with one of the algebras of the class $\mathcal{M}$. Roughly speaking, the program P will be correct w.r.t. the specification $\tilde{S}$ if and only if the algebra defined by P belongs to the class $\mathcal{M}$.[2]

Let us now reexamine the above picture in a modular setting. At one hand we have a **modular** specification $\tilde{S}$ made of some specification modules S_1, S_2,... tied together by some specification-building primitives. On the other hand we have a **modular** program P made of some program modules P_1, P_2,.... Assume moreover that the program structure reflects the specification structure. The problem we have to solve is the following one:

[2]This is of course an oversimplified picture: indeed, the program P should be considered as a correct implementation of $\tilde{S}$ if and only if the algebra defined by P is "behaviorally equivalent" to some algebra belonging to $\mathcal{M}$ (see e.g. [14]). However, in the sequel we shall adopt the oversimplified understanding of program correctness, since it will be sufficient to study the impact of modularity. Note also that our picture does not preclude more refined views about implementations, such as the abstract implementation of one specification by another (more concrete) one [3,7], or the stepwise refinement and transformation of a spe into a piece of software [2]. This indeed is the reason why whe shall speak of "realizations" instead of "implementations".

1. To define a notion of correctness such that "the program module P_2 is correct w.r.t. the specification module S_2" is given a precise meaning, and

2. To ensure that the local correctness of each program module w.r.t. its specification module implies the global correctness of the whole program w.r.t. the whole specification, and

3. To carefully study how some basic requirements about the modular development of modular software, as well as their reusability, interact with the design of the semantics of the (modular) specifications.

It turns out that the main difficulties raised by this goal are twofold:

1. Providing a (loose) semantics to specification modules is not so easy, since from a mathematical point of view (heterogeneous) algebras do not have a modular structure.

2. If our intuition and needs about modular software development and the reuse of modular software can be easily figured out, this is not the case at the level of algebraic semantics.

In the following section we shall try to provide some insight into the solution we propose and into the main ideas underlying what we call the *"stratified loose semantics"*.

3 The stratified loose semantics

For sake of simplicity, we shall focus on the most commonly used specification-building primitive, namely the enrichment one. Moreover, we shall assume that the modular specification we consider is made of one specification module S_2 that enrich only one another specification module S_1, which in turn may enrich other specification modules.

The specification module S_1 determines the specification $\widetilde{S_1}$, the semantics of which is some class of models (or algebras). The signature associated to $\widetilde{S_1}$ is denoted by Σ_1, while the distinguished subset[3] of Σ_1 corresponding to the generators of the defined sorts is denoted by Ω_1. The class of models associated to $\widetilde{S_1}$ is denoted by $\mathcal{M}_1$. Similar notations hold for the S_2 specification module. Note that we have $\Sigma_1 \subseteq \Sigma_2$, and $\Omega_1 \subseteq \Omega_2$. $\mathcal{U}$ denotes the usual forgetful functor from Σ_2-algebras to Σ_1-algebras; the image $\mathcal{U}(\mathcal{M}_2)$ of the class $\mathcal{M}_2$ by the forgetful functor $\mathcal{U}$ will also be denoted by $\mathcal{M}_2|_{\Sigma_1}$, as well as the image by $\mathcal{U}$ of some model M_2 of $\mathcal{M}_2$ is denoted by $M_2|_{\Sigma_1}$.

[3] In Pluss, this distinguished subset is specified apart from the other operations and is introduced by the keyword generated by.

With the help of this simple example, our intuition and needs w.r.t. the modular development of modular software can be summarized as follows:

1. If some piece of software fulfills (i.e. is a correct realization of) the "large" specification $\widetilde{S_2}$, then it must be reusable for simpler purposes (i.e. it must also provide a correct realization of the sub-specification $\widetilde{S_1}$)

2. Any piece of software that fulfills (i.e. that is a correct realization of) the sub-specification $\widetilde{S_1}$ should be reusable as the basis of some correct realization of the larger specification $\widetilde{S_2}$. In other words, it should be possible to implement the sub-specification $\widetilde{S_1}$ without taking care of the (future or existing) enrichments of this specification (e.g. by the specification module S_2).

3. It should be possible to implement the specification module S_2 without knowing which peculiar realization of the sub-specification $\widetilde{S_1}$ has been (or will be) chosen. Thus, the various specification modules should be implementable **independently** of each other, may be simultaneously by separate programmer teams. Moreover, exchanging some correct realization (say P_1) of the specification module S_1 with another correct one (say P_1') should still produce a correct realization of the whole specification $\widetilde{S_2}$, without modification of the realization P_2 of the specification module S_2.

The first two requirements can be easily achieved by embedding some appropriate *hierarchical constraints* into the semantics of the enrichment specification-building primitive. Roughly speaking, it is sufficient to require the following property:

> *Either* $\mathcal{M}_2 = \emptyset$ *(in that case the specification module* S_2 *will be said to be hierarchically inconsistent) or* $\mathcal{M}_2|_{\Sigma_1} = \mathcal{M}_1$.

The third requirement, however, cannot be achieved without providing a suitable (loose) semantics to specification modules. There is no way to take this requirement into account by only looking at the semantics of specifications. The following definition provides the solution we are looking for by embedding the ideas of the initial approach to algebraic semantics into the loose one:

Definition (Stratified loose semantics) :
Let $\mathcal{M}_1$ *be the class of the models of the specification* $\widetilde{S_1}$ *(according to this current definition), and* $\widehat{\mathcal{M}_2}$ *be the class of all the* Σ_2*-algebras finitely generated w.r.t.* Ω_2*, for which the axioms* $Ax(\widetilde{S_2})$ *hold, and which produce* $\widetilde{S_1}$ *models when the new part specified by the specification module* S_2 *is forgotten by the forgetful functor* U *(i.e. we have* $U(\widehat{\mathcal{M}_2}) \subseteq \mathcal{M}_1$*).*

- *If* $\widehat{\mathcal{M}_2}$ *is empty, the enrichment is said to be (hierarchically) inconsistent and the semantics of the specification module* S_2 *is empty, as well as the semantics* $\mathcal{M}_2$ *of the whole specification* $\widetilde{S_2}$.

- *Otherwise, the semantics of the specification module S_2 is defined as being the class $\mathcal{F}_1^2$ of all the mappings $\mathcal{F}_i$ such that:*

  1. *$\mathcal{F}_i$ is a (*total*) functor from M_1 to $\widehat{M_2}$.*
  2. *$\mathcal{F}_i$ is a right inverse of the forgetful functor $\mathcal{U}$, i.e.: $\forall M_1 \in M_1 : \mathcal{U}(\mathcal{F}_i(M_1)) = M_1$.*

  If the class $\mathcal{F}_1^2$ is empty, then the enrichment is also said to be (hierarchically) inconsistent.

- *The semantics of the whole specification $\widetilde{S_2}$ is defined as being the class of all the models image by the functors $\mathcal{F}_i$ of the models of M_1: $M_2 = \bigcup_{\mathcal{F}_i \in \mathcal{F}_1^2} \mathcal{F}_i(M_1)$.*

The class M_2 of the models of the specification $\widetilde{S_2}$ is said to be **stratified** *by the functors $\mathcal{F}_i$.*

Some comments are necessary to better understand the previous definition:

- In the definition above, the restriction to models finitely generated w.r.t. to the generators is made to guarantee that all values will be denotable as some composition of these generators. Thus, structural induction using these generators is a correct proof principle.
- Our semantics is loose, since it associates a class of (non-isomorphic) functors (resp. algebras) to a given specification module (resp. to a given specification). However, our semantics can also be considered as a generalization of the initial approach: under suitable assumptions, the free functor from Σ_1-algebras to Σ_2-algebras is just one specific functor in the class $\mathcal{F}_1^2$.
- It is also important to note that our definition is almost independent of the underlying institution [13].

As a last remark, we must point out how far our definition solves the problem stated in the previous section. A program module will be said to be correct w.r.t. some specification module if and only if it induces a functor belonging to the semantics of the specification module. From our definition, it is then clear that the "composition" of correct program modules (i.e. the program obtained by linking together these program modules) is always a correct realization of the whole specification.

The extension of the definition above to the case where the specification module S_2 enriches more than one specification module as well as its extension to other specification-building primitives (such as e.g. parameterization) do not raise difficult problems and is described in [4].

4 Conclusion

The main significance of the stratified loose framework outlined in this paper is that it is possible to specify and develop software in a modular way, and that the correctness of the implementation should only be established on a module per module basis. A formal theory of software reusability, built on top of our stratified loose semantics, is described in [10].

As a consequence of the "hierarchical constraints" required by modularity, it is necessary to state a careful distinction between "implementable" and "not yet implementable" specification modules. This is done in the **Pluss** algebraic specification language [4,5], the semantics of which is defined following the stratified loose approach. Such a distinction contrasts with all other specification languages developed following either the initial or the loose approach, such as ACT ONE [6,8], ASL [15,1], OBJ2 [9] and LARCH [12], where there is only a distinction between various enrichment primitives.

References

[1] E. Astesiano and M. Wirsing. An introduction to ASL. In *Proc. of the IFIP WG2.1 Working Conference on Program Specifications and Transformations*, 1986.

[2] F.L. Bauer et al. *The Munich Project CIP. Volume I: The wide spectrum language CIP-L.* Springer-Verlag L.N.C.S. 183, 1985.

[3] G. Bernot, M. Bidoit, and C. Choppy. Abstract implementations and correctness proofs. In *Proc. of the 3rd STACS*, pages 236–251, Springer-Verlag L.N.C.S. 210, January 1986.

[4] M. Bidoit. *Pluss, a language for the development of modular algebraic specifications.* PhD thesis, L.R.I., Univ. Paris-Sud, Orsay, France, 1989.

[5] M. Bidoit, M.-C. Gaudel, and A. Mauboussin. How to make algebraic specifications more understandable ? An experiment with the Pluss specification language. *Science of Computer Programming*, 1989. *To appear.*

[6] H. Ehrig, W. Fey, and H. Hansen. *ACT ONE: An algebraic specification language with two levels of semantics.* Technical Report 83-03, Department of Computer Science, TU Berlin, 1983.

[7] H. Ehrig, H. Kreowski, B. Mahr, and P. Padawitz. Algebraic implementation of abstract data types. *Theoretical Computer Science*, October 1980.

[8] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer-Verlag, 1985.

[9] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proc. of the 12th ACM Symposium on Principles of Programming Languages*, pages 52–66, January 1985.

[10] M.-C. Gaudel and Th. Moineau. A theory of software reusability. In *Proc. of ESOP'88*, to appear in Springer-Verlag L.N.C.S., March 1988.

[11] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. *An initial approach to the specification, correctness, and implementation of abstract data types.* Volume 4 of *Current Trends in Programming Methodology*, Prentice Hall, 1978.

[12] J.V. Guttag, J.J. Horning, and J.M. Wing. *Larch in five easy pieces.* Technical Report 5, Digital Systems Research Center, 1985.

[13] D.T. Sannella and A. Tarlecki. Building specifications in an arbitrary institution. In *Proc. of the Intl. Symp. on Semantics of Data Types*, Springer-Verlag L.N.C.S. 173, 1984.

[14] Oliver Schoett. *Data abstraction and the correctness of modular programming.* PhD thesis, University of Edinburg, 1987.

[15] M. Wirsing. *Structured Algebraic Specifications: A Kernel Language.* PhD thesis, Techn. Univ. Munchen, 1983.

Algebraic Concepts for the Evolution of Module Families *

Hartmut Ehrig, Werner Fey, Horst Hansen, Michael Löwe †, Dean Jacobs ‡

April 5, 1989

1 Introduction

The importance of decomposing large software systems into modules to improve their clarity, facilitate proofs of correctness, and support reusablity has been widely recognized within the software engineering community. Recently, considerable interest has developed in techniques for keeping track of structural and historical relationships between modules as a system evolves over time. In this paper, we study these issues within a formal semantic framework for modules based on algebraic specifications. Our goal is to clearly formulate fundamental ideas in this area to serve as a guide to the design of methodologies and tools for software engineering.

We first present an algebraic concept of modules and their interfaces which is suitable for all phases of the software development process; from requirements specification to high-level design specification to executable code. This concept has evolved over the last ten years, from early work on abstract data types [LZ74, GTW76, TWW78], into its present form [WE86, BEPP87]. We then present a set of fundamental operations on interface and module specifications, including horizontal structuring operations for building up specifications, vertical development steps which refine abstract specifications into more concrete forms, and realization of interface specifications by module specifications. A variety of different program development methodologies can be formulated within this framework. For example, a top-down approach might start with high-level requirements expressed as interface specifications. Then, vertical development steps could be taken to elaborate the design, perhaps introducing some horizontal structure. Eventually, the interface specifications would be realized by module specifications to produce a high-level description of the implementation. Finally, additional vertical development steps could be taken until an acceptable implementation is produced.

The algebraic framework allows us to study semantic interactions between horizontal structuring, vertical development, and realization. For example, we study whether horizontal operations are compatible with vertical steps in the sense that a compound module is refined when its submodules are refined. These operations and results concerning their compatibilities are discussed in more detail in [EFH+87].

Our most recent work, discussed here and in more detail in our technical report [EFH+88], studies the construction and evolution of module families. A module family is a collection of conceptually related modules, usually revisions and variants, which have developed over time. Module families provide structure to a module library, facilitating the storage, access, and reuse of its members. In addition, module families allow the members of a group of conceptually related systems to be manipulated all at once rather than individually. In our framework, a module family is defined to be a set of module specifications, each of which realizes a common abstract interface. Each module family has a set of relations, such as *refinement_of*, *revision_of*, and *variant_of*, defined on its members. We show how the horizontal operations on interface and module specifications can be applied to entire module families to produce configuration families and how refinements of the underlying modules induce refinements of configurations.

*This research was carried out as part of an exchange program between TUB and USC.

†TU Berlin, Institut für Software und Theoretische Informatik, Franklinstrasse 28/29, D-1000 Berlin 10

‡CS Dept, University of Southern California, Los Angeles, CA 90089-0782, jacobs@pollux.usc.edu

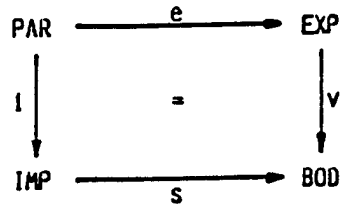


Figure 1: Module Specifications

2 Preliminaries

An algebraic datatype specification is a triple $SPEC = (S, OP, E)$ where S, OP, and E are sets of sort symbols, operation symbols, and equations respectively. A specification morphism $f: SPEC_1 \rightarrow SPEC_2$ between specifications $SPEC_i = (S_i, OP_i, E_i)$ for $i = 1, 2$ is a pair of functions $f = (f_S : S_1 \rightarrow S_2, f_{OP} : OP_1 \rightarrow OP_2)$ such that for each $N : s_1, \ldots, s_n \rightarrow s$ in OP_1 we have $f_{OP}(N) : f_S(s_1), \ldots, f_S(s_n) \rightarrow f_S(s)$ in OP_2, and for each e in E_1 the translated equation $f^{\#}(e)$ is provable from E_2. A $SPEC$-algebra $\mathcal{A}$ consists of a base set A_s for each $s \in S$ and an operation $N_A : A_{s_1}, \ldots, A_{s_n} \rightarrow A_s$ for each operation symbol $N : s_1, \ldots, s_n \rightarrow s$ in OP. The operations are required to satisfy all equations in E. $SPEC$-algebras and homomorphisms between them define a domain $Alg(SPEC)$ used to define the semantics of modules. For each specification morphism $f : SPEC_1 \rightarrow SPEC_2$ there is a forgetful construction $FORGET_f : Alg(SPEC_2) \rightarrow Alg(SPEC_1)$ which forgets all base sets and operations not in $f(SPEC_1)$, and a free construction $FREE_f : Alg(SPEC_1) \rightarrow Alg(SPEC_2)$ which transforms each $SPEC_1$-algebra in $\mathcal{A}_1$ into a freely generated $SPEC_2$-algebra. For more details, see [EM85].

3 Module and Interface Specifications

A *module specification* $MOD = (PAR, IMP, EXP, BOD, i, e, s, v)$ contains four algebraic datatype specifications.

- The *import part* IMP identifies the sorts and operations which are to be brought into into the module. In general, the equations in the import part describe only essential or unusual properties of these operations; their complete definition is left up to the imported module.
- The *export part* EXP identifies those sorts, operations, and equations that are visable outside the module. The export part can be used to hide the representation of data and functions, and to hide auxiliary sorts and operations.
- The *parameter part* PAR contains sorts, operations, and equations which are common to the import and export parts. These components are intended to be generic parameters of the entire modular system and may be instantiated with particular values.
- The *body part* $BODY$ contains equations which define the operations of the export part in terms of the operations of the import part. The body may contain auxiliary sorts and operations which do not appear in any other part of the module.

These algebraic datatype specifications are connected by specification morphisms $i : PAR \rightarrow IMP$, $e : PAR \rightarrow EXP$, $s : IMP \rightarrow BOD$, and $v : EXP \rightarrow BOD$ such that the diagram in figure 1 commutes. The semantics of MOD is given by the function $SEM : Alg(IMP) \rightarrow Alg(EXP)$ mapping import algebras to export algebras as follows: $SEM = FORGET_v \circ FREE_s$. A module specification is said to be *correct* if it is strongly persistent, i.e., if the free construction $FREE_s : Alg(IMP) \rightarrow Alg(BOD)$ leaves the semantics of every import algebra unchanged.

Horizontal structuring operations are used to build up module specifications. The most commonly used horizontal operation is *composition*. The composition of module specifications MOD_1 and MOD_2, denoted $MOD_1 \bullet MOD_2$, connects the import part of MOD_1 with the export part of MOD_2, as shown in figure 2. The connection is established by a pair of specification morphisms $h_1 : IMP_1 \rightarrow EXP_2$ and $h_2 : PAR_1 \rightarrow PAR_2$. The composite module MOD_3 has the same import part as MOD_2, the same export

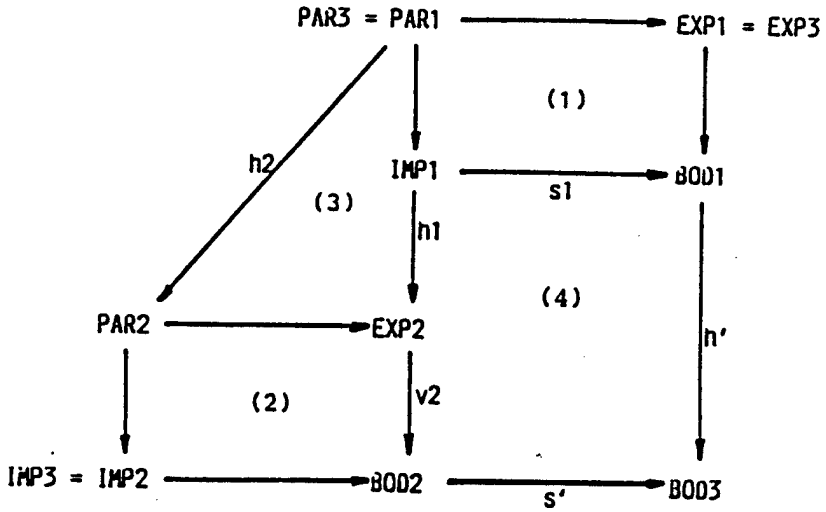


Figure 2: Composition of Module Specifications

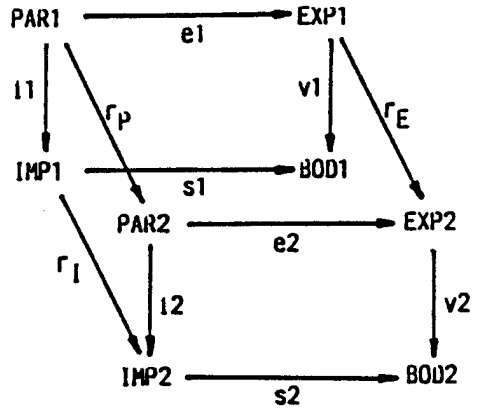


Figure 3: Refinement of Module Specifications

and parameter parts as MOD_1, and body given by the union of the bodies of MOD_1 and MOD_2. The subdiagrams (1), (2), and (3) must commute and (4) is constructed as a pushout diagram. A fundamental result here is that, if MOD_1 and MOD_2 are correct, then MOD_3 is correct and its semantics SEM_3 is given by $SEM_3 = SEM_1 \circ FORGET_{h_1} \circ SEM_3$.

Vertical development steps transform abstract specifications into more concrete forms. The most commonly used horizontal operation is *refinement*. Intuitively, a refined specification more completely describes the resources that the module will produce and the resources that are required to produce them. A refined specification has additional sorts, operations, and equations in its import, export, and parameter parts. A specification and its refined version are connected by three specification morphisms $r_P : PAR_1 \rightarrow PAR_2$, $r_E : EXP_1 \rightarrow EXP_2$, and $r_I : IMP_1 \rightarrow IMP_2$ as shown in figure 3. All subdiagrams in this figure must commute. This is called a *weak* refinement since it satisfies only basic syntactic requirements. A weak refinement is called a refinement if the modules are semantically compatible with respect to their common elements, i.e., if $SEM_1 \circ FORGET_{r_I} = FORGET_{r_E} \circ SEM_2$.

A fundamental result here is that refinement is compatible with composition. Given (weak) refinements MOD_1 by MOD_1' and MOD_2 by MOD_2', and well-defined compositions $MOD_3 = MOD_1 \bullet MOD_2$ and $MOD_3' = MOD_1' \bullet MOD_2'$, there is an induced (weak) refinement of MOD_3 by MOD_3'.

An *interface specification* gives the external features of a module without describing how it is to be implemented. An interface specification $INT = (PAR, IMP, EXP, i, e)$ is simply a module specification without a body $BODY$ and the related morphisms s and v. Horizontal operations, such as composition, and vertical operations, such as refinement, can be restricted to interface specifications. A *realization* of an interface specification is a module specification which implements it. A realization is given by a triple of specification morphisms satisfying the same properties as a weak refinement.

4 Module and Configuration Families

We define a *module family* to be a set of module specifications, each of which realizes a common interface specification. A module family $MODFAM = (INT, (MOD_j, r_j)_{j \in J}, REL)$ consists of an interface specification INT called the abstract interface of $MODFAM$, a family of module specifications MOD_j and realizations $r_j : INT \rightarrow MOD_j$ for each $j \in J$, and a set of relations REL on J. The index set J is assumed to be empty when a new module family is created. An update of the module family entails modification of MOD_j, r_j and the corresponding set J. The set REL is intended to include different relations, such as refinement, between the versions of $MODFAM$.

A *configuration family* is a set of compound modules constructed in a uniform way from a set of module families. Given an n-tuple of module families $MODFAM_i = (INT_i, (MOD_{i_j}, r_{i_j})_{j \in J_i}, REL_i)$ for $i = 1, \ldots, n$, a configuration family $CONFAM = (INT, OP, J, f, REL)$ consists of an interface specification INT called the abstract interface of $CONFAM$, an n-ary horizontal operation OP, a version index set J, an n-tuple of version functions $f = (f_i : J \rightarrow J_i)_{i=1,\ldots,n}$, and a set of relations REL on J. The version functions select n-tuples of module family members to be combined, using OP, to produce members of the configuration family. Each such n-tuple defines one configuration given by some $j \in J$. The corresponding n-tuple is $(MOD^*_{1j}, \ldots, MOD^*_{nj})$ with $MOD^*_{ij} = MOD_{i,f_i(j)}$ for $i = 1, \ldots, n$. Four basic consistency conditions must hold. For example, version consistency makes sure that the tuples of modules can be appropriately combined. We have the following fundamental results.

- *Induced Module Family:* There is an induced module family corresponding to the result of applying OP to those members of $MODFAM_i$ given by the version functions.
- *Induced Refinement:* Refinements between members of the module families induce refinements between corresponding configurations in $CONFAM$ provided that certain basic compatibility conditions hold.
- Induced Updates: Given an update of $MODFAM_i$ by additional realizations, there is an induced update of $CONFAM$ by additional realizations provided that certain basic compatibility conditions hold.

References

[BEPP87] E.K. Blum, H. Ehrig, and F. Parisi-Presicce. Algebraic specification of modules and their basic interconnection. *JCSS*, 34, 1987.

[EFH+87] H. Ehrig, W. Fey, H. Hansen, M. Lowe, and F. Parisi-Presicce. Algebraic theory of modular specification and development. Technical Report 87-06, TU Berlin, 1987.

[EFH+88] H. Ehrig, W. Fey, H. Hansen, D. Jacobs, A. Langen, M. Lowe, and F. Parisi-Presicce. Algebraic specification of modules and configuration families. Technical report, TU Berlin, 1988.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I.* Springer Verlag, Berlin, 1985.

[GTW76] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. Technical Report RC 6487, IBM, 1976.

[LZ74] B. Liskov and S. Zilles. Programming with abstract data types. *SIGPLAN Notices*, 9, 1974.

[TWW78] J.W. Thatcher, E.G. Wagner, and J.B. Wright. Data type specification: parameterization, and the power of specification techniques. In *10th Symp. on Theory of Computing*, 1978.

[WE86] H. Weber and H. Ehrig. Specification of module systems. *IEEE Tran. on Soft. Eng.*, SE-12, 1986.

An Algebraic Approach to the Early Stages of Language Design

Laurette Bradley
Computer Science and Engineering Department
University of California, San Diego
Mail code C-014
La Jolla, California 92093
email : bradley@cs.ucsd.edu

Introduction

We develop a model of the early stages of language design based on universal algebra, and apply this model to a key issue in early design. The benefit of an algebraically based design model is that it allows us to rationalize aspects of design which are otherwise isolated as mysterious processes with little structure. In turn, the benefit of this rationalization is that is allows designs to be made definite, concrete and visible. It is possible to use such designs to reason about design issues. The particular issue to which we apply our model is that of *reusability* of designs: Suppose a language is designed which lacks an important property. How might the language be altered, or *redesigned*, to obtain that property? The particular class of properties that we focus on here concerns the the stages of evaluation of a language (compile time and run time are examples of stages) and the desire to maximize or minimize the amount of evaluation that goes on at each stage. Our claim is that the usual types of language definition and language design tools do not aid the designer in exploring language choices, while ours do.

We have not worked out a model that demystifies every aspect of language design, but we do know how to deal with some important features, and can tell a complete story about abstract representation of "stages of evaluation" in a language definition, based on a notion of removal of information from an algebraic representation of a language. [Bradley 88] and [Bradley 89] give more details of some of the underpinings of the approach that is outlined below. Our goal in this abbreviated paper is to introduce two key notions. One is the notion of an early stage of language design. An early stage is characterized by a lack of syntax, and by wild experimentation with the structure and contents of the basic semantic domain. A late stage, in contrast, is characterized by fairly well accepted syntax, a well understood semantics (informal, at least) and virtually no experimentation with the basic semantic domain. The second notion we introduce is of staged evaluation of an expression in a language. One of the key concerns of a language designer in the early stages is to design a language so that it can be efficiently evaluated. This usually boils down to meaning that efficiency at some stages (such as compile time) will be happily sacrificed to improve efficiency at other stages (such as run time). This paper outlines a way to examine and manipulate stages of evaluation early in the formal design of a language.

The early stages of design (informal)

Unlike syntax and semantics, design is not an aspect of artificial languages that has been amenable to formalization. Most studies of the design of programming languages have been extremely informal ([Ghezzi 87], [MacLennan 87], [Hoare 73], [Wirth 74], for example). Typically, these treatments are advice on the properties a language should have, such as orthogonality and readability. More formal approaches to language design, range from early work on extensible languages to very recent work on semantics based design and tools for language specification ([Paulson 82], [Lee 87], [Blikle 86] and [Ligler 75]). These approaches are uniformly based on the idea that language design begins when a syntax is formally defined, and finishes when a notion of semantics is made precise. More useful are approaches such as Pratt's ([Pratt 83]) describing signifcant paradigm shifts in the design of languages, and approaches that deal with the design of constructs to solve particular problems, such as [Hudack

88], and [Solworth 88].

At a more practical and realistic level language design begins with the design of the semantic domain -- the decision on what is to be represented in the language. This decision requires insight, imagination, and deep understanding of the particular domain. In the earliest stages of design language designers are not faced with issues such as whether the constructs in the language are orthogonal or readable. These issues are so generic that advice on them can not help a designer faced with the particular problems of a specific domain. Some of the realistic questions of the early design stage are the following. A designer might

- want to develop language constructs for highly parallel operations on list. (See [Solworth 88], for example)
- want to design a language in which software faults are reduced. This requires a thorough understanding of how software faults occur, followed by a design of constructs to avoid them.
- feel that it is necessary to allow some type cheating in a language, but might also want to restrict it. The designer might want to develop a construct for "structured" type cheating, but may be unsure how the structuring should be done. (see [Geschke 77] for a discussion of this in Mesa.)
- want to allow a wide variety of notions for arrays, from static, as in Pascal, to fully dynamic, as in APL, but may not understand the affects this decision will have on the compilation and run time of the programs.

The typical, extremely general, advice given in informal treatments of language design is not useful in these sorts of design situations, and neither are the semantics based language design tools, since they force the designer to start by giving a syntax, and these issues are prior to that stage.

The early stages of design (formal)

The model of language design that we present here rationalizes design by presenting it as *definitional*, *informative* and *concrete*. By *definitional* we mean that the formal expression of the design can be used to define the language. By *informative* we mean that information about the language that is lacking in other forms of definition, for example syntax and semantics is present in the formally expressed design. By *concrete* we mean designs done in *steps* so that they have stable intermediate results, and so that the method of going from one stable intermediate to another is describable and computable.

The model of language design can be summed up as follows. All language design starts with a *world*, which has *structure*. Informally, a world is anything that can be symbolized, and its structure is a classification of the types of objects, functions and relations in the world. Formally, a world is a many-sorted algebra, and its structure is identified with the signature of the algebra.

Language design proceeds as a sequence of decisions on how to represent the objects, functions and relations of the world. In the early stages of design the crucial step is acquiring new world views by manipulating world structure. The variation in world views may be extreme -- for instance replacing horn clauses without equality by equational logic in the logic underlying a relational language such as prolog -- or it may be minor -- choosing a new name for an object. In our model of design each of these manipulations is a mapping from algebras to algebras. The final result is a design which is expressed as a sequence of such mappings. Designs rationalized in this manner are then open for inspection and manipulation.

Our model of design is based on three kinds of manipulations. These are

- *Metaphor* or *Structure adoption*. This is the most radical kind of manipulation that can be performed on a world, and is loosely analogous to the working of metaphor in natural language, where the structure of one domain is adopted to structure another domain. For example, the linear ordering on temperatures, can be used to order the domain of putters, as in "His putting is hot", in which the implication is that his

putting is very "high", or good. With respect to language design, the following are examples of structure adoption: A construct, such as a new repeat statement, could be added to the language from another language. A more complex example, which happens quite often, is to adopt structure within the language from its semantic domain. For example, the semantics of a *read* statement might involve checking the next input value to ensure that its type is compatable with the type of the variable being read to, and invoking an error continuation if it is not. Adopting this action of the runtime system into the language itself is very useful, and amounts to allowing programmer control over exception handling for *read*.

- *Structure reshaping*. In this manipulation the structure of the world is altered, but in such a way that all the computations expressible in the original world, or some subset thereof, are expressible in the altered world. This kind of manipulation is exemplified by activities such as forming a derived operation over the original algebra, merging separate operations into one, or renaming an operation.
- *Splitting*. In this manipulation the structure of the world is represented jointly by two or more separate worlds. This manipulation allows the language designer to introduce stages of evaluation into the language, as will be described more below. The intuition behind this is that the designer can control the stage at which information about objects becomes available by removing them, or parts of them, from the original algebra and placing them in associated algebras which are available at another stage of evaluation.

A particular problem concerning "stages of evaluation"

We can apply this formalization of the early stages of language design to an issue which is central to design. The issue is this: Suppose we have a language L, defined by grammar G and semantics S, which does not satisfy some property P. What changes we can make to L (that is, to G and S) so that the changed language L', defined by grammar G', and semantics S', satisfies P? This is clearly a central issue in design; it is also quite vast. We examine instances of this question for a class of properties that capture aspects of time of evaluation for expressions in a program. In particular we restrict our attention to the solution of this problem for properties that concern the "stage of evaluation" at which certain information is known about a program. For example the property of being statically typed -- i.e. the property that all types can be determined at compile time -- is such a property. We will now discuss the idea of stages of evaluation more generally.

What are stages of evaluation?

The usual way of thinking about language evaluation is *timeless*. An expression in a language is given a meaning by the semantic function which maps expressions into meanings and their is no notion of stages in this evaluation. Yet in both natural and artificial languages there are many examples of languages whose semantics are given in *stages*. For example, in natural language semantics the classical treatment of intension and extention is a strategy for separating the "meaning" of a sentence in the abstract -- the intension, given as a function from possible worlds to truth values -- from the "value" of a sentence as used in a particular situation -- the extension, given as a truth value. In modern unification-based treatments, such as Head-driven phrased structure grammar ([Pollard 87]) a similar recognition of the stages of evaluation for natural languages is expressed in the treatment of the meaning of a term as coming about in a cummulative fashion via the interaction of constraints arising from several sources (phonological, syntactic, semantic, contextual).

Artificial languages too, have this same aspect. In particular, the evaluation of a program in a programming language can be viewed very naturally as coming about in a cummulative fashion via the interaction of information about its evaluation that is gathered at various stages. For example, a classical instance of this is the representations of finite mappings in programming languages. Most languages support the same notion of finite mapping, as arrays, but they differ widely in the constraints they place on the stages of evaluation at which information about the finite mapping has to be known. In Pascal all information about the finite mapping, including all dimension, bounds and component type, has to be present in the text of the program itself; in AlgolW all information about dimension and component type has to be present in the program text, but the information about the bounds can be delayed until the execution of the prologue to the block in which the array was declared is executed; finally, in a language such as

APL no information about the dimension, bounds, or component type has to be fixed in the program text. Rather, all of this information can be supplied *repeatedly* at run-time.

This notion of stages of evaluation is important because many aspects of the efficiency of program evaluation are tied to it. Abstractly, when evaluation of a program occurs in stages it is typical that these stages are not viewed uniformly. It is much more important for some stages to be performed quickly, or using less space, than it is for others. Each stage uses resources of time or space differently, and it is often a key goal of language evaluation to shift activities from one stage to an earlier or later one so as to improve the efficiency of a critical stage. Not only does this discussion characterize the distinction of "compile-time" (where time efficiency is usually not critical) versus "run-time" (where time efficiency is often critical), but it also characterizes the various stages of compile-time itself: the ordering of the activities of optimization, intermediate code generation, and register allocation is often critical to the efficiency of the compiler, and to its ability to perform some activities at all.

How do stages of evaluation show up in a semantics? In a denotational semantics they do not show up at all. In other kinds of semantics, for example VDM, they show up very concretely as fixed times (compile-time, run-time) with respect to which program evaluation has to be expressed. One of our goals is to develop a style of writing semantics so that the staged evaluation of a language can be expressed naturally. Also, so that the properties of the computation that occur at each stage (time and space requirements, for example) can be analyzed.

What is the significance of stages of evaluation for language design? Given that a language designer has a desire to represent some given world in a language, and given constraints about what information must be known at various stages of evaluation, or what activities must happen at various stages, what are a language designers choices? Clearly, the designer must attempt to design a language so that the constraints on the stages of evaluation are satisfied.

Given the model of design that we propose, we can describe the range of languages that could be used as languages for some underlying world of interest. Also, we can show how a language that lacks some property can be can be redesigned to acquire it.

The designer's choices: How stages of evaluation are manipulated in designs

The goal of the current work is to be able to express the important practical phenomena of stages of evaluation in terms of *removal of information from an algebra representing a language or world.* Also, we want to be able to reason about solutions to problems in languages concerning stages of evaluation. The removal of information from an algebra forces the need for a later stage of evaluation in which the information is presented. There are two important aspects of this. One is that when information is removed from an algebra the meanings of the remaining objects have to change. Intuitively, if one designs a language to represent some world, one would expect that the meanings of the constructs in the language would have to change in proportion to the difference between the structure of the world and the structure of the language. In the case of splittings the meanings of the constructs left behind are functions of the information removed. The second important aspect is that evaluation that took place at one stage before might now be shifted to another stage, and that this shift might be unacceptable for reasons of efficiency.

In the case that the shift of evaluation to another stage is unacceptable, the language designer has two choices. One is to simply undo the design step that introduced the shift of parts of the evaluation to the new stage. The other, more interesting alternative, is to attempt to incorporate into the language a version of the actions that have been delayed to a later stage. Algebraically, this amounts to raising semantic operations to the level of the language. Although it was, of course, designed by completely different methods than the ones we propose, the Algol68 variant case statement provides an excellent example of the one of the kinds of constructs that emerges from this strategy of incorporating late stage semantic activities into the language itself in our design model. This statement forces the programmer to explicitly code for the checking of the current type of the variant, and to provide statements to be executed in the case of any possible outcome of this check. This assures that type correctness of the program can be determined at compile time, even though the presence of variants for which value and type information is delayed until runtime would seem to preclude that.

[Blikle 87] Blikle, A., "Denotational Engineering or from Denotations to Syntax", in *VDM '87: VDM - F Formal Method at Work*, Springer-Verlag, 1987.

[Bradley 88] Bradley, L., "A Treatment of Languages with Stages of Evaluation", Proceedings of the third workshop on Mathematical Foundations of Programming Language Semantics, pp. 425 - 443, Springer-Verlag, 1988.

[Bradley 89] Bradley, L., "An Algebraic Approach to the Early Stages of Language Design", *Full paper in preparation*

[Geschke 77] Geschke, C., Morris, J., Satterthwaite, E., "Early Experience with Mesa", in *Communications of the ACM*, August 1977, v. 20, no. 8, 1977.

[Ghezzi 87] Ghezzi, C. Jazayeri, M., *Programming Language Concepts 2nd edition*, John Wiley and sons, 1987.

[Gopinath 89] Gopinath, K. Hennessy, J., "Copy Elimination in Functional Languages", Proceedings ACM Symposium on Principles of Programming Languages, 1989.

[Hoare 73] Hoare, C. A. R., "Hints on Programming Language Design", Keynote address given a ACM SIGACT/SIGPLAN Conference on Principles of Programming Language. Boston: October, 1973. See also, Stanford University Computer Science Dept., Technical Report STAN-CS-74-403.

[Mou 88] Mou, Z., and Hudack, P., "An Algebraic Model for Divide-and-Conquer and its Parallelism", *The Journal of Supercomputing*, 2, pp 257-278, 1988.

[Lee 87] Lee, Peter and Pleban, Uwe. "A Realistic Compiler Generator Based on High-Level Semantics", Proceedings ACM Symposium on Principles of Programming Languages, pp 284-295, 1987.

[Ligler 75] Ligler, G. T., "A mathematical approach to language design", Proceedings ACM Symposium on Principles of Programming Languages, 1975, p41-53.

[MacLennan 87] MacLennan, B. J., *Principles of Programming Languages: Design, Evaluation, and Implementation*, Holt, Rinehart and Winston, 1987.

[Paulson 82] Paulson, L., "A semantics-directed compiler generator", Proceedings ACM Symposium on Principles of Programming Languages, pp 224-233, 1982.

[Pratt 86] Pratt, V., "Five Paradigm Shifts in Programming Language Design and their Realization in Viron, a Dataflow Programming Environment" Proceedings ACM Symposium on Principles of Programming Languages, pp 1-9 1983.

[Solworth 88] Solworth, J., "Programming Language Constructs for Highly Parallel Operations on Lists", *The Journal of Supercomputing*, 2, pp 331-347, 1988.

[Wirth 74] Wirth, N., "On the design of programming languages", appearing in Information Processing 74, North-Holland publishing company, 1974.

Algebraic methods in programming language theory

Carolyn Talcott
Stanford University
CLT@SAIL.STANFORD.EDU

1. Overview

The point of this paper is to describe a challenging application of algebraic methods to programming language theory. Much work on the theory of Scheme–like languages (applicative, but not necessarily functional) has an essentially algebraic flavor [Talcott 1985] [Felleisen, Wand, et.al. 1988]. Thus it seems appropriate to make the algebraic aspect explicit. This would allow us to take advantage of the work in algebraic methods to extend and generalize existing work and to facilitate application of the results. Full support of this application of algebraic methods will require bringing diverse results together in a single enriched framework.

The goal of our work is to develop a general semantic framework that provides a formal basis and tools for a wide range of programming activities such as: design and implementation of languages; dynamic language extension; building programming environment tools; specifying programs, including programs that operate on other programs; proving properties of programs; and program transformation, including compiling, high-level optimizations, partial evaluation, programming-in-the-large, and program derivation.

To support such a range of activities it is necessary to support a variety of programming paradigms and to provide many views of programs: programs as data to construct, transform, and annotate; programs as descriptions of computation to execute and analyse; and programs as black boxes distinguished only by observable behavior. To effectively use the various views of programs one also needs formal connections relating them.

An algebraic setting provides a unifying framework for the various views of programs. The use of syntactic algebras, data algebras, and algebras of computation states provides a uniform treatment of data, textual, and control abstraction mechanisms in languages. The semantic equivalence relations induced by models of a specification correspond to a generalization of the notion of *comparison relation* [Talcott 85] with equivalence in terminal models being the maximal such equivalence. Placing our work in an algebraic setting also increases the potential for cross fertilization with other approaches such as abstract actions [Mosses 84] and the categorical view of computation [Moggi 89].

---

† This research was partially supported by DARPA contract N00039-84-C-0211

2. Applicative languages

We start from Landin's view of programming languages as enriched versions of the lambda calculus [Landin 66]. Particular languages are determined by choices for abstract computation states, primitive data, and primitive operations to enrich the basic mechanisms of naming, abstraction, and application. Within this framework we can treat a variety of programming primitives including functional abstractions, control abstractions, objects with memory, dynamic environments, and reflection mechanisms. We study notions of program equivalence, formal systems for proving program equivalence, tools for compiling and transforming, derived computations (non-standard interpretations) such as cost of execution, reference count, strictness, trace, and abstract interpretations.[1] To illustrate some of the issues we will outline the kernel of this family of languages and discuss various extensions, and refinements.

2.1. The kernel

The language consists of expressions **Exp** generated from given sets of variables **Var** and constants **Con** by application and abstraction. **Ve** is the set of value expressions — variables, constants, and abstractions.

$$\mathbf{Ve} = \mathbf{Var} + \mathbf{Con} + \lambda\mathbf{Var}.\mathbf{Exp} \qquad \mathbf{Exp} = \mathbf{Ve} + \mathrm{app}(\mathbf{Exp}, \mathbf{Exp})$$

We adopt the convention that exp, exp_0, ...range over **Exp**, ve, ve_0, ...range over **Ve**, and similarly for other syntactic and semantic domains. The basic semantics is given in terms of computations states and transitions. The semantic domains include values (**Val**), environments (**Env**), continuations (**Cnt**), and states (**St**). Values include constants and closures of lambda expressions $\langle\lambda var.exp, env\rangle$. Environments are finite maps from variables to values. Continuations are stack like objects that describe the rest of the computation. States are tuples with at least a local component and a continuation component. The local component is either an expression-environment pair or a value.

$$\mathbf{Val} \supseteq \mathbf{Con} + \langle\lambda\mathbf{Var}.\mathbf{Exp}, \mathbf{Env}\rangle \qquad \mathbf{Env} = [\mathbf{Var} \xrightarrow{\mathrm{f}} \mathbf{Val}]$$

$$\mathbf{Cnt} = \{\mathtt{top}\} + \mathtt{appi}(\mathbf{Exp}, \mathbf{Env}, \mathbf{Cnt}) + \mathtt{appc}(\mathbf{Val}, \mathbf{Cnt})$$

$$\mathbf{St} = \langle\mathbf{Exp}, \mathbf{Env}, \mathbf{Cnt}, \ldots\rangle + \langle\mathbf{Val}, \mathbf{Cnt}, \ldots\rangle$$

Transitions are defined by the single step relation $\mapsto$. The rules for application are:

$$\langle \mathtt{app}(exp_0, exp_1), env, cnt\rangle \mapsto \langle exp_0, env, \mathtt{appi}(exp_1, env, cnt)\rangle$$

$$\langle val, \mathtt{appi}(exp_1, env, cnt)\rangle \mapsto \langle exp_1, env, \mathtt{appc}(val, cnt)\rangle$$

$$\langle val, \mathtt{appc}(\langle\lambda var.exp, env\rangle, cnt)\rangle \mapsto \langle exp, env\{var := val\}, cnt\rangle$$

---

[1] For examples see [Talcott 85,86], [Mason 86], [Felleisen 87], [Mason and Talcott 89a,b].

2.2. Adding primitive operations

We extend the kernel language to treat such programming primitives as abstract data types (natural numbers, lists, ...), control abstractions, objects with memory, dynamic binding, and reflection mechanisms. For example to treat objects with memory we assume memory operations such as `mk`, `get`, `set` are among the constants. We add cells to the value domain, a memory component to states, and rules for applying memory operations.

$$\mathbf{Val} \supseteq \mathbf{Cel} \qquad \mathbf{Mem} = \mathbf{Cel} \xrightarrow{\mathrm{f}} \mathbf{Val}$$

$$\mathbf{St} = \langle \mathbf{Exp}, \mathbf{Env}, \mathbf{Cnt}, \mathbf{Mem}, \ldots \rangle + \langle \mathbf{Val}, \mathbf{Cnt}, \mathbf{Mem}, \ldots \rangle$$

$$\langle val, \mathrm{appc}(\mathtt{mk}, cnt), mem \rangle \mapsto \langle cel, cnt, mem\{cel := val\} \rangle \qquad \text{\% if } cel \notin \mathrm{Dom}(mem)$$

The rules for application are as before since the memory component is unchanged by these transitions.

2.3. Denotations and program equivalence

To define evaluation we introduce an answer domain **Ans** and an operation Unload mapping final states $\langle val, \mathtt{top}, \ldots \rangle$ to answers. A program context *cxt* is an environment together with the non-local components of a state. The evaluator Ev maps expressions and program contexts to answers and is defined by $\mathrm{Ev}(exp, cxt) = ans$ if $\langle exp, cxt \rangle \overset{*}{\mapsto} st$ for some final state *st* such that $\mathrm{Unload}(st) = ans$, where $\overset{*}{\mapsto}$ is the transitive reflexive closure of $\mapsto$. From this definition we can derive the usual equational definition of a denotational interpreter. We can then abstract on the semantic domains to admit a wider class of models. The denotation of an expression is then a partial function mapping program contexts to answers.

By a program equivalence relation we mean an equivalence relation on expressions. We use several notions of program equivalence. An operational equivalence relation is determined by a set of program contexts and a notion of indistinguishability of answers. Two expressions are operationally equivalent if in all relevant contexts they give indistinguishable answers. Contexts can be either semantic contexts as above or expressions with holes. A denotational equivalence relation is determined by a class of models. Two expressions are denotationally equivalent if they have the same denotation in all models under consideration. A program equivalence may also be characterized as the least or greatest equivalence relation satisfying some closure conditions. For example a reduction calculus is determined by a set of reduction rules and the induced equivalence is the congruence closure of the reduction rules.

2.4. Intensions

As a tool for studying intensional aspects of computation we introduce the notion of derived computations. Let **Dval** be a domain of derived values. Derived states are state-derived value pairs. A derived computation is determined by a derivor map $D \in [\mathbf{St} \times \mathbf{St} \times \mathbf{Dval} \to \mathbf{Dval}]$. Rules for transitions on $\mathbf{St} \times \mathbf{Dval}$ are obtained from the basic transition rules by defining $\langle st, dval \rangle \mapsto \langle st', D(st, st', dval) \rangle$ if $st \mapsto st'$. A derived evaluator Dev is obtained from a derived computation by specifying a derived answer domain **Dans** and a derived unloading operation $\mathrm{Dunld} \in [\mathbf{St} \times \mathbf{Dval} \to \mathbf{Dans}]$. Then

Dev($exp, cxt, dval$) = $dans$ if <<exp, cxt>, $dval$> $\overset{*}{\mapsto}$ <$st, dval'$> for some final st such that Dunld($st, dval'$) = $dans$. As with the standard evaluator we can abstract from the state transition definition and also provide a basis for developing computable approximations. Reference counting and cost analyses can be explained by derived computations. Several examples of derived computations are worked out in [Talcott 86].

If we want to reason about occurrences of expressions we can replace expressions by labels together with a map *fetch* from labels to a pair consisting of a tag and a label sequence. A tag is either app, λ_{var}, a constant, or a variable and the label sequence labels subexpression occurrences. Transition rules are modified accordingly. For example if $fetch(lab) = (\mathtt{app}, [lab_0, lab_1])$ then <lab, env, cnt> $\mapsto$ <$lab_0, env, \mathtt{appi}(lab_1, env, cnt)$>.

3. Towards an algebraic theory

To make the algebraic aspects of our theory explicit we work with programming language algebras (PL algebras). Following [Broy, et. al. 1987] our PL algebras specify syntactic and semantic entities in a single (partial) algebraic theory. The theory has a kernel which is elaborated and refined in various ways. In the algebraic setting these can all be thought of as operations on theories. Some operations change the theory while some only change the presentation. Most operations are naturally determined by local features. Operations illustrated above include: adding new semantic domains, adding summands to domain equations, adding components to structures, restructuring — replacing states by expression-context pairs or replacing expressions by locations plus the *fetch* map, addition of transition rules, and lifting of transition rules on enriched states.

To study program equivalence we need mechanisms for specifying classes of program contexts, notions of indistinguishability of answers, and classes of models. We also need mechanisms for handling reduction rules, for expressing closure operations such as congruence, transitive, and equivalence, and more generally for forming least or greatest relations satisfying certain conditions. We also need tools for reasoning with and about the resulting relations based on the form of definition. One goal of our generalized algebraic framework is to obtain a deeper understanding of operational equivalence by examining richer classes of observing contexts. Thus it will be of interest to consider non–reachable models and families of models parameterized by classes of primitive operatations.

To provide tools for program analysis we need tools for abstracting and encapsulating various levels of specification, for instantiating to particular interpretations, for refining an interpretation, and for relating different interpretations. This suggests treating abstractions of specifications and descriptions of particular interpretations as first class objects with tools for doing "algebra–in–the–large". In many cases we need to focus attention on particular models by specifying additional axioms and model–theoretic constraints such as initiality, finality, reachablility. Thus a formal language for expressing some class of model–theoretic constraints would be of great help.

Finally we will want to embed PL algebras into mechanized reasoning systems to facilitate semantics based formal reasoning about programs. In particular we will want the ability to express and reason about general first order or even higher order properties. This is a challenge both to algebraic methods and to builders of mechanized reasoning to systems to make natural embeddings possible.

4. References

Broy, M., Winsing, M., and Pepper, P. [1987] On the algebraic definition of programming lanuguages, *ACM TOPLAS*, **9(1)**, pp. 54–99.

Felleisen, M. [1987] The calculi of lambda-v-cs conversion: A syntactic theory of control and state in imperative higher-order programming languages, Ph.D. thesis, Indiana University.

Felleisen, M., Wand, M., Friedman, D. P., and Druba, B. F. [1988] Abstract continuations: a mathematical semantics for handling full functional jumps, *Proceedings 1988 ACM conference Lisp and functional programming*, pp. 52–62.

Landin, P. J. [1966] The next 700 programming languages, *Comm. ACM*, **9**, pp. 157–166.

Mason, I. A. [1986] The semantics of destructive Lisp, Ph.D. Thesis, Stanford University. CSLI Lecture Notes No. 5, Center for the Study of Language and Information, Stanford University.

Mason, I. A. and Talcott, C. L. [1989a] A Sound and Complete Axiomatization of Operational Equivalence between Programs with Memory, (LICS89).

Mason, I. A. and Talcott, C. L. [1989b] Programming, Transforming, and Proving with Function Abstractions and Memories, (ICALP89).

Moggi, E. [1989] Computational lambda-calculus and monads (LICS 89).

Mosses, P. [1984] A basic abstract semantic algebra, in: *Semantics of data types, international symposium, Sophia-Antipolis, June 1984, proceedings*, edited by G. Kahn, D. B. MacQueen, and G. Plotkin, Lecture notes in computer science, no. 173 (Springer, Berlin) pp. 87–108.

Talcott, C. [1985] The essence of Rum: A theory of the intensional and extensional aspects of Lisp-type computation, Ph.D. Thesis, Stanford University.

Talcott, C. [1986] Rum: An intensional theory of function and control abstractions, in: Boscarol, M. Aiello, L. C., Levi, G. (eds.) *Workshop on Functional and Logic Programming, Trento Italy, Dec 1986*, Lecture Notes in Computer Science **306** (Springer-Verlag).

Dynamic Extensions of Programming Language Semantics

Ilie Parpucea
University of Cluj-Napoca
Str. M. Kogălniceanu Nr. 1
3400 Cluj-Napoca, Romania

The increasing requirement for flexibility and efficiency of various complex programming applications demand the new programming languages to be extensible. The existing language extension methods allow only static extensibility of programming languages. These methods are restricted to be static by language implementation by means of syntax-oriented compilers and hence, they do not allow dynamic changes (i.e. adaptation or extension of the language according to the real needs of the language user).

The dynamic extension of programming languages is not a new problem. However, its actual solution and implementation on specific cases have not been fully explored. This is due to the fact that the language extension is very complex and difficult to apply in the environment of language specification by grammar and syntax-directed compiler implementation controlled by derivations using the specification grammar. We will sketch in this paper a model for language extensibility based on the dynamic extension of the language semantics in an environment in which language specification rules are interpreted as operation schemes of an algebra rather then rewriting rules of a context-free grammar.

The mathematical machinery providing support for the design of programming language semantics is the HAS hierarchy [Rus83]. The HAS hierarchy allows the creation of a formal mechanism for the specification of the concept of an *hierarchical abstract computing system*. The objects that belong to such an abstract computing system are represented as formal expressions which are organized into an algebra of words $\mathcal{W}$ [Gra68], [Pur77] and can be constructed dynamically following a hierarchy of layers[Rus83], each layer being constructed on top of the previous layers of the hierarchy. Therefore, the concept of an abstract computing system is considered here as the mathematical support for dynamic specification of the semantics of a programming language. It is is specified by means of a hierarchy of heterogeneous algebras.

A heterogeneous algebra is a triple

$$\mathcal{A} = \{D, \Sigma S, F\}$$

where D is the set of primitive and composed computing objects, ΣS is the operation scheme set, consisting of primitive and composed operation schemes, while F is the function which associate to each operation scheme $\sigma \in \Sigma S$ a computing operation. The assumption is that the carrier D of the algebra $\mathcal{A}$ is a family of sets $D = \{D_i | i = 0, 1, \ldots n\}$ and the operation schemes in ΣS can be organized into a hierarchy $\Sigma S =$

$\{HAS(0), HAS(1), \ldots, HAS(p)\}$ such that if $\sigma \in HAS(0)$ then $F(\sigma)$ is a *nullary-operation*, that is, $F(\sigma)$ is a constant in a set $D_j, 0 \leq j \leq n$. For each $i > 0$, if $\sigma \in HAS(i)$, $F(\sigma)$ has as the domain a direct product of sets used as ranges of the operations associated with the operation schemes in $HAS(i-1), \ldots, HAS(0)$ and as the range a set already used as the range of operations in $HAS(i-1), \ldots, HAS(0)$, or a new set of D not yet used as the range of any other operation.

The computation behavior of each operation associated with the operation schemes as shown above is specified by a collection of specific formal identities.

The objects of the abstract computing system are represented as formal expressions organized into a heterogeneous algebra of words

$$\mathcal{W}(\mathcal{V}) = \{V, \Sigma S, F\}$$

where V is a set of symbols used to denote constants of the computing system and for each $\sigma \in \Sigma S$, $F(\sigma)$ is a rule of word formation under the restrictions specified above for the algebra $\mathcal{A}$.

The notion of *semantics dynamic extensibility* is expressed by the dynamic character of the algebras $\mathcal{A}$ and $\mathcal{W}$. It allows dynamic definition of new operations in $\mathcal{A}$ which supply new dynamic expression forms in $\mathcal{W}$. Since $\mathcal{A}$ and $\mathcal{W}$ are two similar algebraic structure, their dynamic extensibility are related by homomorphisms $f : \mathcal{A} \rightarrow \mathcal{W}$ and $e : \mathcal{W} \rightarrow \mathcal{A}$ such that $f = e^{-1}$ and $e = f^{-1}$. The construction of these homomorphisms can be sketched as follows: Since $\mathcal{W}$ is an initial algebra of the class of algebras specified by ΣS there exists a unique homomorphism $h : \mathcal{W} \rightarrow \mathcal{A}$ that coincides with a function $e : V \rightarrow D$ on the generator set V. On the other hand any surjective function defined on the free generators of the carrier of the semantic algebra and taking values in the free generators of the algebra $\mathcal{W}$ can uniquely be extended to an homomorphism. It can be shown that this homomorphism is an inverse of the homomorphism obtained by extending e to the homomorphism $e^{\#} : \mathcal{W} \rightarrow \mathcal{A}$ that evaluates the free generators of $\mathcal{W}$ to the values they denote and conversely. Moreover, this property is preserved by dynamically extending the original heterogeneous algebras $\mathcal{A}$ and $\mathcal{W}$ by taking their carriers as index sets of the family of sets supporting a new level of heterogeneous algebras. Since the carrier of a programming language algebra has a finite set of generator classes, this construction can be used to put together the syntax algebra and the semantics algebra of a programming language into a a programming language specified by a pair of algebras related as above and to organize them into a hierarchy of layers. Thus, the process of dynamic extension of the expression forms of an algebra can directly be applied to the dynamic extension of the programming language semantics. An application of this result is shown in the context of Clear specification language in [Bur80]. We illustrate this application developing a semantics model of a Pascal-like programming language expressed in terms of its representation in a machine meant to implement it. This is done by layering of the language algebras on the following levels:

- Let D_0 be the set of primitive data types of a programming language. This set can

be organized as an algebra

$$\mathcal{A}_l = \{D_0, \Omega_0, F_0 : \Omega_0 \rightarrow I\}$$

where Ω_0 is the set of symbols denoting nullary-operations defined on the carrier set of the primitive data types, F_0 is the function that associates to each $\sigma \in \Omega_0$ its memory representation length in standard units (bytes, words, etc), while I is a subset of natural numbers. The set I will be taken as the index set of the next level of the language hierarchy.

- The level 1 of the language hierarchy is defined using the level 0 as the selector set for the domain of the operations on level 1 and has the form:

$$\mathcal{A}_1 = \{D_1 = \{D_i | i \in I\}, (\Sigma S_o)_{o \in \Omega_0}, F_0 : D_1 \rightarrow I, F_1\}$$

where:

  - D_1 represents a partitioning of D_0 into classes of data types, the partitioning criterion being the representation length.
  - ΣS_o is the set of operation schemes providing the definition of the new types of objects in terms of objects of level 0.
  - F_0 is a function specifying the domain and the range of all operation schemes of the new operations while F_1 is the function that associates the new operation schemes with computation rules.

- In order to define the level 2 of the language hierarchy one must consider the manner of data interpretation. This is performed by considering the following algebra specifying the semantics of level 2:

$$\mathcal{A}_2 = \{D_2 = \{D_{i,j} | i \in N, j \in M\}, (\Sigma S_{o_{i,j}}), F_2 : D_1 \times M \rightarrow N \times M, F_2'\}$$

where:

  - $D_{i,j}$ represents the carrier of the i-th length data type interpreted in the j-manner.
  - $\Sigma S_{o_{i,j}}$ is the set of operations schemes of the new language level.
  - F_2 specifies the index set of the carrier D_2 of the level 2 of the language algebra while F_2' is the function that associates each operation scheme $o \in \Sigma S_{o_{i,j}}$ and a j-manner of interpretation with a heterogeneous operation specific to the level 2 of the hierarchy.
  - The set M contains possible manners of interpretation while the set N contains possible interpretation lengths.

- The level three of the language algebra hierarchy is specified by

$$\mathcal{A}_3 = \{D_3 = \{D_i | i \in I\}, (\Sigma S_i), F_3'\}$$

  which allows the definition of certain type constructors for the introduction of the Pascal like data types **record, file, set**. This level of the language algebra hierarchy will preserve the carrier of the preceding level and will enrich it with new operations characteristic to the newly defined data types.

The mathematical machinery developed in [Rus83] under the name of Heterogeneous Algebraic Structures, HAS-hierarchy, models the construction of the new types of objects and allows the dynamic extension of a language algebra on a practical unlimited number of layers. The construction of the new layers of computing objects supported by the language depends only on the types of objects required by the application and the imagination and the ability of its constructor. The application of this mathematical machinery for language development allowing dynamic language extensibility according to the language user computing needs is shown in [Rus88].

References

[Bur80] Burstall, R.M., Goguen, J. A., "The Semantics of Clear a Specification Language", Proceedings of 1979 Copenhagen Winter School on Abstract Software Specification, pages 292-332, Springer-Verlag 1980.

[Gra68] Grãtzer, G., *Universal Algebras*, Van Nostrand, Princeton, 1986.

[Pur77] Purdea, I., Pic, G., (in Romanian), *Tratat de Algebră Modernă*, Vol. 1, Ed. Acad. R.S.R., Bucureşti, 1977.

[Rus83] Rus, T., (in Romanian), *Mecanisme Formale Pentru Specificarea Limbajelor*, Monograph, Ed. Acad. R.S.R., Bucureşti, 1983.

[Rus88] Rus, T., "Parsing Languages by Pattern Matching", *IEEE Transactions on Software Engineering*, 14:4, pp. 498-510, April 1988.

General Logics*

José Meseguer
SRI International, Menlo Park, CA 94025, and
Center for the Study of Language and Information
Stanford University, Stanford, CA 94305

The main question addressed in this talk is:

What is a logic?

that is, how should general logics be axiomatized? The talk, based on a recent paper of mine[1], proposes a specific axiomatic answer to this question and applies that answer to obtain axioms for logic programming.

Beyond their application to logic programming, the axioms given here for a logic are sufficiently general to have wide applicability within logic and computer science. The connections between these two fields are growing rapidly and are becoming deeper. Besides theorem proving, logic programming, and program specification and verification, other areas showing a fascinating mutual interaction with logic include type theory, concurrency, artificial intelligence, complexity theory, databases, operational semantics and compiler techniques. The concepts presented in this talk are motivated by the need to understand and relate the many logics currently being used in computer science, and by the related need for new approaches to the rigorous design of computer systems. Therefore, this work has goals that are in full agreement with those of J.A. Goguen and R. Burstall's theory of institutions; however, it addresses proof-theoretic aspects not addressed by institutions. In fact, institutions can be viewed as the model-theoretic component of the present theory. The main new contributions include a general axiomatic theory of entailment and proof, to cover the proof-theoretic aspects of logic and the many proof-theoretic uses of logic in computer science; they also include new notions of mappings that interpret one logic (or proof calculus) in another, an axiomatic study of categorical logics, and the axioms for logic programming.

*Supported by Office of Naval Research Contracts N00014-82-C-0333 and N00014-86-C-0450, NSF Grant CCR-8707155 and by a grant from the System Development Foundation.

[1] "General Logics" in: H.-D. Ebbinghaus et al. (eds.) Proc. Logic Colloquium'87, North-Holland, 1989.

LOTOS: An Algebraic Specification Language for Distributed Systems

Luigi Logrippo
University of Ottawa
Computer Science Department, Protocols Research Group
Ottawa, Ont. Canada K1N 9B4
e-mail: lmlsl@acadvm1.uottawa.ca

1. Background

The theory and practice of specification languages for data communications protocols and services (often called Formal Description Techniques or FDTs) has been the object of much recent interest. Formal and exact specifications of protocols and services are useful in every phase of the protocol development life-cycle. Even more, they are essential for protocols and services that are international standards, meant to be implemented in compatible ways across the world. The specification must capture those features of an implementation that are necessary for it to be able to communicate with other implementations. Therefore, it is important that the specification be precise and implementation-independent.

The International Organization for Standardization (ISO) has been developing over the years a family of standardized data communications protocols, called OSI (Open Systems Interconnection). At the very beginning of this effort it was recognized that, in order for OSI to be a real standard, it was necessary to provide it with an appropriate FDT, in which OSI standards could be specified. An international committee (of which the author of this paper is a member) set out to produce such a standard FDT, and, some years later, the language LOTOS has now become an International Standard [ISO]. Interestingly enough, the language is turning out to be very appropriate not only for OSI protocols and services, but also for a wide family of distributed systems. In this paper, we intend to offer a very brief overview of the basic philosophy of LOTOS and of research work being carried out around it. Much additional information on LOTOS can be found in [VVD][ISO], and in the annual series of Proceedings *Protocol Specification, Testing and Verification*, published by North Holland.

2. LOTOS Principles

LOTOS, the Language of Temporal Ordering Specifications, is one of the most precisely defined languages in use today. Its static semantics are defined by an attributed grammar, while its dynamic semantics are based on algebraic concepts. LOTOS is made up of two components: a *data type* component, which is based on the algebraic specification language ACT ONE [EM], and a *control* component, which is based on a clever mixture of Milner's CCS [M] and Hoare's CSP [H]. Most of the theoretical framework of the control component, and especially the concept of *internal action* are based on Milner's work. In particular, non-determinism is modelled by internal actions as in [M] rather than by adding special operators as in [H]. The rendez-vous semantics follow Hoare's "multi-way rendez-vous" concept, by which all processes that share a gate must participate in a rendez-vous on that gate. Actions, however, can be transformed into internal actions by *hiding* them. In this way, further participation in the action of processes outside the *hide* is prevented.

LOTOS dynamic semantics for the *control* component is expressed in operational terms by inference rules as in [M], and the operators were chosen in such a way that it has been possible to prove about them a rich set of algebraic properties, similar to those of [M]. Therefore, the language is at the same time "executable" (by virtue of the operational semantics), and amenable to proof techniques (by virtue of the algebraic properties).

The language is purely recursive in nature, without side effects. It supports process parameterization, where it is possible to specify both value and gate parameters.

Some of the most important operators of the *control* part are: [] (choice), |[A]| (parallel execution with synchronization via gates in set A), || (parallel execution with synchronization on *all* gates),

||| (parallel execution in interleave), **hide** (hiding of gates), >> (sequential composition of processes), and [> (*disable*, modelling a nondeterministic interruption).

The *data* part supports parameterized types, type renaming, and conditional rules.

Because of the fact that LOTOS is made up of what its designers viewed as the most valid parts of CCS and CSP, the language has considerable expressive power. It favors a highly structured specification style and top-down, as well as bottom-up, design. For example, following some ideas already present in [H], "constraint-oriented" specifications are possible in LOTOS, i.e. a specification can be designed as a collection of processes each one of which imposes its own logical constraints on the overall system behavior (this turns out to be a powerful way to impose "separation of concerns"). Other styles, useful for different purposes (e.g., implementation specification, state-oriented specification, etc.) are also possible, and a theory of how to transform a specification style into another is being developed.

3. Executability of LOTOS Specifications and LOTOS Tools

Because of the fact that LOTOS is (partially) executable, a specification is effectively a "fast prototype" of the entity specified, thus it is possible to exercise a specification of a complex system at the design stage. This means that design errors can be found much earlier in the software development cycle than with other techniques.

The two LOTOS interpreters in existence today are described in [L][GHL][VVD].

4. Verification in LOTOS

It is possible to carry out in LOTOS proofs such as the ones found in [M][H], and the proof methods are similar to those found in these references. The best developed proof techniques involve the concept of "bisimulation" [P][B1]. Proof methods based on the concepts of "traces" and "refusal sets" [H] are also being considered. Unfortunately however, because of the presence of internal actions, some of the proof methods developed for CSP, such as fixpoint induction methods, do not seem to be applicable to LOTOS.

An important open problem is to find a unified verification framework for both the *control* and the *data* part.

Of course, the challenging aspect is to be able to prove properties of systems of realistic size. To this end, computer-assisted verification tools are being envisioned.

5. A Theory of Implementation and Testing

A rich formal theory of implementation and testing is being developed around LOTOS [BB][B]. This means that the relation "I is an implementation of S" is formally defined for two expressions I and S. This formalization is given by the *reduction* relation, where I *reduces* S if: i) I can only execute actions that S can execute and: ii) I can only refuse actions that can be refused by S. Intuitively, I can be more deterministic than S, and can contain fewer options. In other words, in LOTOS the abstraction of a specification with respect to the implementation is represented by a higher level of nondeterminism.

Similarly, the relation *A and B are testing equivalent* [DH] has been formally defined as: *A reduces B and B reduces A*. Roughly speaking, two specifications are testing equivalent if their externally observable behaviors are identical. This corresponds to the *failure equivalence* of Hoare [H]. By using these concepts, it is possible to derive implementations and test cases in a formal way from a LOTOS specification.

It must be observed, however, that so far these concepts have been fully developed for restricted forms of the language only.

6. LOTOS in Practice

Specifications of real-life systems of thousands of lines have been written in LOTOS. Some of these are on their way towards becoming part of ISO International Standards. Some examples

are: several OSI layers (Network, Transport, Session), specifications of telephone systems [FLS], etc. (in addition of course to all best known "textbook" examples such as the Alternating Bit Protocol, the Dining Philosopher's problem, etc.). Several such examples are included in [VVD]. The language is starting to be used in industrial environments, and the results appear to be quite promising.

7. A LOTOS Example

The following example, adapted from [BB], is a LOTOS specification for an entity which is able to accept three natural numbers in any order and stops after printing the largest of them.

```
01      specification Max3[in1,in2,in3,out] : noexit

02      type integer is
03           sorts int
04           opns
05            zero      :            -> int
06            succ      :      int   -> int
07            largest   :   int,int  -> int
08           eqns forall X,Y: int ofsort int
09            largest ( zero , X   ) =  X;
10            largest (  X , zero ) =  X;
11            largest ( succ(X) , succ(Y) ) = succ ( largest(X,Y) );
12      endtype

13          behavior
14               hide mid in
15                 (
16                     Max2 [in1,in2,mid]
17                   |[mid]|
18                     Max2 [mid,in3,out]
19                 )

20               where
21                     process Max2 [val1,val2,max] : noexit :=
22                          (  val1?X:int; exit(X, any int)
23                             |||
24                             val2?Y:int; exit(any int, Y)
25                          )
26                                >> accept V: int, W: int in
27                                       max!largest(V,W); stop
28                     endproc
29      endspec
```

The specification is to be read as follows:

Lines 2 to 12 define the type *integer* with its associated operation *largest*. This is done according to the semantics of [EM]. Of course, the standard LOTOS library contains all these definitions, so normally the user will include them by invoking the library.

Lines 14 to 19 describe the top structure of the specification, which consists of two instantiations of process *Max2*. The latter is capable of finding the largest of two numbers, read in any order from gates *val1* and *val2*, and outputting it on gate *max*. As the two copies of *Max2* are instantiated, their gates are renamed respectively *in1, in2, mid,* and *mid, in3, out*, resulting in the fact that the output value computed by one copy is fed to the other over gate *mid*. Note that *mid* is

hidden, because it is meant for internal communication between the two instances of *Max2* only.

Lines 21 to 28 describe process *Max2*. It allows interleaving between the input actions on gates *val1* and *val2*. Both values input are then forwarded to the action on line 27, which calculates the largest of them and inputs it.

Lines 22 to 27 could also be written as follows:

```
  val1?X:int; val2?Y; max!largest(X,Y); stop
[]
  val2?Y:int; val1?X; max!largest(X,Y); stop
```

and the equivalence between the two specifications could be proved easily by using the simplest rules of bisimulation.

Acknowledgment Work reported here was funded in part by the Natural Sciences and Engineering Research Council of Canada, the Telecommunications Research Institute of Ontario, the Department of Communications, and Bell-Northern Research. Acknowledgment is of course also due to the work of the colleagues of the LOTOS group, and especially to its two successive chairmen, Chris Vissers and Ed Brinksma. We are indebted to Souheil Gallouzi for useful comments on an earlier version of this paper.

References

[BB] Bolognesi, B., and Brinksma, E. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems 14 (1987)* 25-59. Also reprinted in [VVD].

[B] Brinksma, E. A Theory for the Derivation of Tests. In [VVD], pp. 235-247.

[B1] Brinksma, E. On the Design of Extended LOTOS. PhD Thesis, Twente University (NL), 1988.

[DH] DeNicola, R., and Hennessy, M.C.B. Testing Equivalences for Processes. *Theoretical Computer Science 34, (1984),* 83-133.

[EM] Ehrig, B., Mahr, B. *Fundamentals of Algebraic Specifications.* Springer-Verlag, 1985.

[FLS] Faci, M., Logrippo, L., and Stepien, B. Formal Specification of Telephone Systems in LOTOS. To appear in: Brinksma, E., Scollo, G., and Vissers, C. (eds.) *Protocol Specification, Testing, and Verification IX,* North-Holland.

[GHL] Guillemot, R., Haj-Hussein, M., and Logrippo, L. Executing Large LOTOS Specifications. In: Aggarwal, S., and Sabnani, K. (eds.) *Protocol Specification, Testing, and Verification VII,* North-Holland, 1988, 399-410.

[ISO] International Organization for Standardization. Information Processing Systems. Open Systems Interconnection. LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behavior (ISO International Standard 8807), 1988.

[L] Logrippo, L., Obaid, A., Briand, J.P., and Fehri, M.C. An Interpreter for LOTOS, a Specification Language for Distributed Systems. *Software-Practice and Experience, 18 (1988)* 365-385.

[H] Hoare, C.A.R. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[M] Milner, R. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, 1980.

[P] Park, D. Concurrency and Automata on Infinite Sequences, Proc. 5th GI Conference, *Lecture Notes in Computer Science N. 104,* 167-183, 1981.

[VVD] van Eijk, P., Vissers, C.A., and Diaz, M. *The Formal Description Technique LOTOS.* Nort-Holland, 1989.

Modeling Distributed Systems as Distributed Data Types

Steven P. Miller
Jon G. Kuhl
Department of Computer Science
Department of Electrical and Computer Engineering
University of Iowa, Iowa City, Iowa

INTRODUCTION

One of the most difficult problems in the design and verification of distributed systems is the development of an appropriate notion of abstraction. In the last few years, the *process algebra* approaches of Milner[1], Hennessy[2-3], DeNicola[4-5], Bergstra and Klop[6], and Brookes, Hoare, and Roscoe[7] have made substantial progress in this area. This paper explores the use of many concepts from the theory of process algebras within the more traditional framework of abstract data types [8-9].

Distributed data types (DDTs) arise naturally in a variety of situations in which a distributed system can be viewed a single object. The distributed nature of such objects manifests itself through spontaneous, internal operations which may alter the object's externally visible behavior. Formally, we define a DDT as a heterogeneous algebra supplemented by such internal operations. In our approach, a distributed system is first *specified* as a single DDT, then *modeled* as a family of distributed objects (which are in turn specified as DDTs) and processes which act upon these objects. *Verification* consists of showing that the model correctly implements the specification by offering only behaviors that are more deterministic than those allowed by the specification.

This procedure may be repeated for each distributed data type used in the model, allowing stepwise refinement of the specification to any level of detail. At each step, the complexity of analysis (e.g., state space explosion) is controlled through elimination of internal operations which do not alter the observable behavior of the object.

DISTRIBUTED DATA TYPES

We first extend the traditional definition of a signature [8-9] to accommodate the notion of objects which may alter their behavior through spontaneous internal operations.

Definition 1 An S-sorted distributed signature $\Sigma =< S, F, I >$ consists of

- a set S of *sort* names.
- a family $F_{w,s}$ of sets of *external* operation names, where $w \in S^*$ and $s \in S$. For convenience, $f \in F_{w,s}$ is depicted as $f : s_1 \times \ldots \times s_n \rightarrow s$ where $w = s_1, \ldots, s_n$, and F is taken to be $\cup_{w,s} F_{w,s}$.
- a family I_s of sets of *internal* operation names, where $s \in S$. Again, $i \in I_s$ is depicted as $i : s \rightarrow s$ and I is taken to be $\cup_s I_s$.

Example 1 The signature of a distributed, bounded queue of length 3 may be given as $\Sigma =< S, F, I >$ where $S = \{queue, item\}$ and

$$\begin{array}{lllllllllll} F = & \{ & nq: & & \rightarrow & queue, & enq: & queue \times item & \rightarrow & queue, \\ & & deq: & queue & \rightarrow & queue, & next: & queue & \rightarrow & item\} \\ I = & \{ & \tau_1: & queue & \rightarrow & queue, & \tau_2: & queue & \rightarrow & queue\} \end{array}$$

In what follows, it will be useful to compare signatures which differ only in their internal operations.

Definition 2 Let $\Sigma =< S, F, I >$ and $\Sigma' =< S', F', I' >$ be S and S' distributed signatures. Then $\Sigma \subseteq \Sigma'$ (read Σ *is contained by* Σ') if $S = S'$, $F = F'$, and $I_s \subseteq I'_s \; \forall s \in S$.

A particular DDT is specified as a heterogeneous Σ-algebra supplemented with spontaneous internal operations. As in the process algebra approach [1-7], interactions between a DDT and its environment are *synchronous* in that a DDT may *refuse* to participate in an inappropriate operation. Refusal of an operation is indicated by $\oslash$, and we adopt the convention that refusals propagate, i.e., $f(a_1, \ldots, a_n) = \oslash$ if $a_i = \oslash$ for any $1 \leq i \leq n$.

Definition 3 Let $\Sigma =< S, F, I >$ be an S-sorted distributed signature. Then a Σ-distributed data type (Σ-DDT) A consists of:

- a set A_s for each $s \in S$ (called the *carrier* of A of sort s).
- an *external* function $f^A : A_{s_1} \times \ldots \times A_{s_n} \rightarrow A_s \cup \{\oslash\}$ for each $f \in F_{w,s}$ where $w = s_1, \ldots, s_n$.
- an *internal* function $i^A : A_s \rightarrow A_s \cup \{\oslash\}$ for each $i \in I_s$.

Example 2 Let ITEM be a predefined set of objects, $u \in ITEM$, and $x, y, z \in ITEM \cup \{\lambda\}$, where λ is a special symbol not in ITEM representing the absence of an item. A particular DDT, A, for the signature of Example 1 then consists of the carriers $A_{queue} = \{< x, y, z > |\ x, y, z \in ITEM \cup \{\lambda\}\}$, $A_{item} = ITEM$ with operations

$$
\begin{array}{rcll}
nq^A(\,) & = & < \lambda, \lambda, \lambda > & \\
enq^A(< x, y, z >, u) & = & < u, y, z > & if\ x = \lambda,\ \oslash\ otherwise \\
deq^A(< x, y, z >) & = & < x, y, \lambda > & if\ z \neq \lambda,\ \oslash\ otherwise \\
next^A(< x, y, z >) & = & z & if\ z \neq \lambda,\ \oslash\ otherwise \\
\tau_1{}^A(< x, y, z >) & = & < \lambda, x, z > & if\ x \neq \lambda\ and\ y = \lambda,\ \oslash\ otherwise \\
\tau_2{}^A(< x, y, z >) & = & < x, \lambda, y > & if\ y \neq \lambda\ and\ z = \lambda,\ \oslash\ otherwise
\end{array}
$$

BEHAVIORS

As described earlier, verification of a distributed system consists of showing that the DDT generated by a model of the system correctly implements the DDT given as the specification. To define this more rigorously, we introduce the concept of a *behavior*.

Definition 4 Let $X = \{x_1, \ldots, x_n\}$ be a set and let $X_\oslash = X \cup \{\oslash\}$. Then

1. If $x \in X_\oslash$ then $\{x\}$ is a behavior of X with $root(\{x\}) = x$ and $succ(\{x\}) = \emptyset$.
2. If $x \in X_\oslash$ and $\beta_1, \ldots, \beta_n$ are $(0 \leq n < \infty)$ distinct behaviors of X, then $\{x, \beta_1, \ldots, \beta_n\}$ is a behavior of X with $root(\{x, \beta_1, \ldots, \beta_n\}) = x$ and $succ(\{x, \beta_1, \ldots, \beta_n\}) = \{\beta_1, \ldots, \beta_n\}$.

Example 3 If $X = \{a, b, c, d\}$ then $\{\oslash\}, \{a\}, \{a, \{b\}\}$, and $\{\oslash, \{a, \{\oslash\}\}, \{b\}\}$ are behaviors of X, while $\{\,\}, \{a, b\}$, and $\{a, \{\oslash, b\}\}$ are not behaviors of X.

Behaviors are simply trees in which some of the nodes may be a refusal, $\oslash$, and define how the result of an operation may change over time. For example, $\{\oslash, \{false\}, \{true\}\}$ (read *refusal, eventually false or true*), describes the behavior of an operation which is initially refused but must eventually return either the value *false* or *true*. In this example, the refusal is referred to as a *transient* behavior while *false* and *true* are referred to as *stable* behaviors. Behaviors may also be *finite* or *infinite*. In what follows, we will consider only finite behaviors.

Definition 5 Let β be a behavior of X. Then β is *stable* if $succ(\beta) = \emptyset$ and *transient* if $succ(\beta) \neq \emptyset$. β is *finite* if β is stable or if $\forall b \in succ(\beta), b$ is finite.

A behavior β *implements* a behavior β' if it is more deterministic, i.e., if every stable behavior of β is a stable behavior of β' and no transient behavior of β contradicts β'.

Definition 6 Let β and β' be finite behaviors of X. Then $\beta \sqsubseteq \beta'$ (read β implements β') if any of the following are true

1. $succ(\beta) = succ(\beta') = \emptyset$ and $root(\beta) = root(\beta')$
2. $succ(\beta) \neq \emptyset$ and $root(\beta) = root(\beta')$ and $\forall b \in succ(\beta)$, $b \sqsubseteq \beta'$
3. $succ(\beta) \neq \emptyset$ and $root(\beta) = \oslash$ and $\forall b \in succ(\beta)$, $b \sqsubseteq \beta'$
4. $\exists b' \in succ(\beta')$ such that $\beta \sqsubseteq b'$.

Implementation is a preorder (a relation which is reflexive and transitive) over behaviors and naturally induces an equivalence relation (the *kernel* of $\sqsubseteq$) over behaviors [3]. Two behaviors are said to be *equivalent* if they implement each other.

Definition 7 Let β and β' be finite behaviors of X. Then $\beta \sim \beta'$ (read β is equivalent to β') if $\beta \sqsubseteq \beta'$ and $\beta \sqsupseteq \beta'$.

Example 4 $\{a\} \sim \{a\}$, $\{b\} \sqsubseteq \{a,\{b\}\}$, $\{a\} \not\sqsupseteq \{a,\{b\}\}$, $\{\emptyset,\{a\}\} \sim \{a\}$, $\{a,\{a,\{b\}\}\} \sim \{a,\{b\}\}$, $\{a,\{\emptyset,\{a\},\{b\}\},\{b\}\} \sim \{a,\{a\},\{b\}\}$, $\{a,\{\emptyset,\{a\},\{b\}\},\{b\}\} \not\sqsupseteq \{a,\{a\},\{b,\{a\}\}\}$

IMPLEMENTATIONS OF DDTs

In one sense, an element of a carrier of a DDT possesses not only the capabilities explicitly defined for it, but also those of all objects into which it may evolve through its internal operations. Accordingly, we associate each object with the behavior consisting of itself and those objects into which it may evolve, formally given in Definition 8 as the behavior returned by the τ operator. Note that such behaviors contain no refusals. If every behavior generated by τ for a DDT is finite (stable), then the DDT is said to be finite (stable).

Definition 8 Let A be a $\Sigma =< S, F, I >$ DDT and let $a \in A_s$. Then

$$\tau_s^A(a) = \{a\} \cup \{\tau_s^A(i^A(a)) \mid i \in I_s \wedge i^A(a) \neq \emptyset\}$$

If $\tau_s^A(a)$ is finite (stable), a is finite (stable). If $\tau_s^A(a)$ is finite (stable) $\forall a \in A_s$, A is said to be finite (stable) for sort s. If A is finite (stable) $\forall s \in S$, A is said to be finite (stable).

Example 5 In the DDT of Example 2,

$$\begin{aligned}
\tau^A_{queue}(< x,\lambda,\lambda >) &= \{< x,\lambda,\lambda >\} \cup \{\tau^A_{queue}(< \lambda,x,\lambda >)\} \\
&= \{< x,\lambda,\lambda >\} \cup \{\{< \lambda,x,\lambda >\} \cup \{\tau^A_{queue}(< \lambda,\lambda,x >)\}\} \\
&= \{< x,\lambda,\lambda >\} \cup \{\{< \lambda,x,\lambda >\} \cup \{\{< \lambda,\lambda,x >\}\}\} \\
&= \{< x,\lambda,\lambda >\} \cup \{\{< \lambda,x,\lambda >, \{< \lambda,\lambda,x >\}\}\} \\
&= \{< x,\lambda,\lambda >, \{< \lambda,x,\lambda >, \{< \lambda,\lambda,x >\}\}\}.
\end{aligned}$$

This DDT is finite but not stable.

Applying an operation to behaviors also results in a behavior, as described in Definition 9. Note that behaviors obtained as the result of an operation may contain refusals.

Definition 9 Let A be a finite $\Sigma =< S, F, I >$ DDT, $f : s_1 \times \ldots \times s_n \rightarrow s \in F$, and β_i be behaviors of A_{s_i}, $1 \leq i \leq n$, such that $\beta_i = \{b_i, \Gamma_{i,1}, \ldots \Gamma_{i,n_i}\}$. Then

$$f^A(\beta_1,\ldots,\beta_n) = \{f^A(b_1,\ldots,b_n)\} \cup \{f^A(\beta_1,\ldots,\beta_{i-1},\Gamma_{i,j},\beta_{i+1},\ldots,\beta_n) \mid 1 \leq i \leq n, 1 \leq j \leq n_i\}$$

Example 6 Consider the distributed queue DDT of Examples 2 and 5.

$$\begin{aligned}
next^A(\tau^A_{queue}(< x,\lambda,\lambda >)) &= next^A(\{< x,\lambda,\lambda >, \{< \lambda,x,\lambda >, \{< \lambda,\lambda,x >\}\}\}) \\
&= \{next^A(< x,\lambda,\lambda >)\} \cup \{next^A(\{< \lambda,x,\lambda >, \{< \lambda,\lambda,x >\}\})\} \\
&= \{\emptyset\} \cup \{\{next^A(< \lambda,x,\lambda >)\} \cup \{next^A(\{< \lambda,\lambda,x >\})\}\} \\
&= \{\emptyset\} \cup \{\{\emptyset\} \cup \{\{next^A(< \lambda,\lambda,x >)\}\}\} \\
&= \{\emptyset\} \cup \{\{\emptyset\} \cup \{\{x\}\}\} \\
&= \{\emptyset\} \cup \{\{\emptyset,\{x\}\}\} \\
&= \{\emptyset,\{\emptyset,\{x\}\}\}
\end{aligned}$$

Example 7 If $\beta_1 = \{a,\{b\},\{c\}\}$ and $\beta_2 = \{d,\{e\}\}$, then (omitting the intermediate steps)

$$f(\beta_1,\beta_2) = \{f(a,d),\ \{f(b,d),\{f(b,e)\}\},\ \{f(c,d),\{f(c,e)\}\},\ \{f(a,e),\{f(b,e)\},\{f(c,e)\}\}\}$$

Traditionally, one data type is said to be implemented by another if there exists a homomorphism from the implementation to the specification. We extend this notion to DDTs by defining a homomorphism from behaviors of the implementation to behaviors of the specification.

Definition 10 Let A be a finite Σ^A-DDT and B be a finite Σ^B-DDT such that $\Sigma^B \subseteq \Sigma^A$. Then an *implementation homomorphism* $\Phi : A \rightarrow B$ is a family of functions $< \Phi_s : A_s \rightarrow B_s >_{s \in S}$ such that

for all $f : s_1 \times \ldots \times s_n \rightarrow s \in F$ and all $< a_{s_1}, \ldots, a_{s_n} > \in A_{s_1} \times \ldots \times A_{s_n}$

$$\Phi_s(f^A(\tau^A_{s_1}(a_{s_1}), \ldots, \tau^A_{s_n}(a_{s_n}))) \sqsubseteq f^B(\tau^B_{s_1}(\Phi_{s_1}(a_{s_1})), \ldots, \tau^B_{s_n}(\Phi_{s_n}(a_{s_n})))$$

where $\Phi_s(\oslash) = \oslash \forall s \in S$. If Φ preserves $\sim$ rather than $\sqsubseteq$, then Φ is said to be an *equivalence homomorphism.*

The existence of an implementation homomorphism guarantees that an implementation can exhibit only behaviors which are more deterministic than the behaviors allowed by the specification. If A implements B, then A may be safely substituted for B in any larger context. An equivalence homomorphism guarantees that an implementation can exhibit all the behaviors allowed by the specification, and vice-versa. If A is equivalent to B, then A may be substituted for B and B may be substituted for A in any larger context.

Example 8 Let A be the distributed queue DDT of Examples 1 and 2. Let B be a stable queue of length 3 (i.e., a DDT with no internal operations) such that B has the same signature as A except that $I_{queue} = \emptyset$. Let the carriers of B be $B_{queue} = \{w \mid w \in ITEM^*, \mid w \mid \leq 3\}$ and $B_{item} = ITEM$, and the operations of B be

$$\begin{array}{rcll} nq^B() & = & \lambda & \\ enq^B(w,u) & = & uw & if \mid w \mid < 3, \oslash otherwise \\ deq^B(w) & = & w' & if\ w = w'u, \oslash otherwise \\ next^B(w) & = & u & if\ w = w'u, \oslash otherwise \end{array}$$

where λ is the empty string, $w \in ITEM^*$ such that $\mid w \mid \leq 3$, and $u \in ITEM$. Then $\Phi_{queue}(< x, y, z >) = xyz$ and $\Phi_{item}(u) = u$ is an equivalence homomorphism from A to B. For example, making use of the results of Example 6 we have

$$\begin{array}{rcl} \Phi_{item}(next^A(\tau^A_{queue}(< x, \lambda, \lambda >))) & = & \Phi_{item}(\{\oslash, \{\oslash, \{x\}\}\}) \\ & = & \{\oslash, \{\oslash, \{x\}\}\} \\ & \sim & \{x\} \\ & = & \{next^B(x)\} \\ & = & next^B(\{x\}) \\ & = & next^B(\tau^B_{queue}(x)) \\ & = & next^B(\tau^B_{queue}(\Phi_{queue}(< x, \lambda, \lambda >))) \end{array}$$

Similar results hold for all possible operations and arguments, establishing that a distributed queue of length 3 is equivalent to a stable queue of length 3. In verifying any larger system incorporating a distributed queue, this allows us to substitute the stable specification to simplify the analysis.

BIBLIOGRAPHY

[1] R. Milner, *A Calculus of Communicating Systems*, LNCS 92, Springer-Verlag, New York, 1980.

[2] M. Hennessy and R. Milner, "Algebraic Laws for Nondeterminism and Concurrency", *JACM*, Vol. 32, No.1, pp. 137-161, 1985.

[3] M. Hennessy, *Algebraic Theory of Processes*, The MIT Press, Cambridge, Massachusetts, 1988.

[4] R. de Nicola and M. Hennessy, "Testing Equivalences for Processes," in *Proc. ICALP'83*, LNCS 154, pp. 548-560, 1983.

[5] R. de Nicola and M. Hennessy, "CCS without τ's," in *Proc. TAPSOFT'87*, LNCS 249, pp. 138-152, 1987.

[6] J. Bergstra and J. Klop, "Process Algebra: Specification and Verification in Bisimulation Semantics", *Mathematics and Computer Science II*, CWI Monograph 4, North-Holland, Amsterdam, 1986.

[7] S. Brookes, C. Hoare, and A. Roscoe, "A Theory of Communicating Sequential Processes", *JACM*, Vol. 31, No.3, pp. 560-599, 1984.

[8] J. Gougen, J. Thatcher, and E. Wagner, "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," *Current Trends in Programming Methodology*, Prentice Hall, New Jersey, 1978.

[9] S. Kaplan and A. Pnueli, "Specification and Implementation of Concurrently Accessed Data Structures: An Abstract Data Type Approach," in *Proc. STACS'87*, LNCS 247, pp. 220-244, 1980.

A FORMAL MATHEMATICAL MODEL FOR DETECTING SUBROUTINE DEPENDENCES : A LOGIC PROGRAMMING APPROACH

Dan Ionescu, Lawrence Wen
University of Ottawa,Department of Electrical Engineering, 770 King Edward,
Ottawa,Ontario, K1N 6N5, Canada,
email: diopb @ uottawa.BITNET

Abstract: In this paper the problem of sequential Fortran restructuring is considered. The need of reusing the large amount of scientific programs written in sequential Fortran captured the attention of various computer scientists since the arrival of parallel computers. This problem needs a good abstract approach in order to provide an intelligent software package which can automatically execute the task.. In this paper the case of subroutine dependences is considered. A formal mathematical model based on the discrete event system theory is first introduced. Futher results are obtained b\ased on this model. The recurrence property of the model suggested to approach the implementation through a logic programming technique. An expert system shell was used to easy the implementation. Practical results and a demonstration package resulted.

1 .INTRODUCTION

The increasing interest in parallel computers and their capabilities to speed-up the execution of computational intensive scientific programs generated an accrued research to support programmers with more enhanced programming tools.

Scientific programming characterized by a high-level of floating-point computation usually uses Fortran programs to implement a requested algorithm. A lot of already available sequential code must be reconsidered and rewritten, or in other words restructured [1], in order to be executed in a parallel environment.

There are commonly available computers [1], [2], [5] which vectorize the code written in standard Fortran. The compiler attempts to convert the innermost loops to vector operations.

Even though great progress has been made in automatic code restructuring, the only automatic system available to date is limited to individual loops [1]. The analysis of parallelism in independent nested DO loops has been also reported [8]. Parallelism at a larger granularity must be explicitly specified by the programmer [1].

In this respect, programming environments that could help a programmer to develop explicitly parallel programs are or have been under research [1], [3] for specific architectures.

In order to efficiently use a parallel multiprocessor system it is necessary not only to achieve the fine-grain parallelism, through the DO loop vectorization, but also the coarse-grain parallelism such as subroutine calls.

However, this subject remained untouched because of the complexity of the analysis process for the parallelism detection.

When affirming this we have in our mind the case of multiple levels of subroutine calls which is obvious in any reasonable and well structured FORTRAN program. As an enforcement of the last statement we mention the Cray-1 FORTRAN compiler which stops the vectorization when a subroutine call is encountered [1].

In order to achieve this fine analysis the compiler or other software that can do this, has to be provided with reasoning capabilities [1]. This means unification, reasoning mechanism, forward chaining, backtracking a.s.o.

The development of logic programming techniques provides this environment which allows the computer to deal with problems requiring intelligence. A practical implementation of this piece of software will be finally in the form of an expert system.

Expert system shells that can interface external libraries in Fortran are excellent environments that provide a lot of facilities to accomplish the task of parallelizing sequential FORTRAN code.

The blackboard of an expert system shell provides the storage elements which help to solve this problem dynamically. On the other hand, the powerful reasoning capabilities of the expert system provide other necessary complex mechanisms for obtaining the subroutine dependences.

An example a Fortran program containing an arbitrary number of subroutines is processed by the expert system.

2. PRELIMINARIES

In order to provide a mathematical approach to the detection of subroutine dependences the following notations will be introduced:

- the set of input variables: $U = \{ u_k \mid u_k$ real and integer variables,strings, arrays,etc., k N$\}$
- the set of output variables: $Y = \{ y_k \mid y_k$ real and integer variables,strings, arrays,etc., k N$\}$
- the set of commands: $C = \{ c_k \mid l_1$: if e go to l_2 else go to l_3, fi
  V l_1: $(x_1, x_2, ..., x_n) := (t_1, t_2, ..., t_n)$,go to l_2 $(n \geq 1)\}$

where l_i are labels l_1= end, l_2=l_3,e a quantifier free formula, and fi is the finish if

- the set of elementary " subprograms" $S_o = \{ s_k \mid s_k$ elementary functions or subroutines$\}$,where an elementary function or subroutine is considered that function or subroutine which does not call any subprogram.
- the set of all functions or subroutines $S \quad S_o$

Under this consideration a subroutine is defined as follows:

$$s : U \times C \times P \longrightarrow Y$$

The dependence relations are introduced as follows:

Definition 1: Consider i<j in a lexicographical order; the subroutine $s_j(u_1,...,u_{nj},y_1,...,y_{mj})$ is said to be dependent on subroutine $s_i(u_1,...,u_{ni},y_1,...,y_{mj})$ if one of the following conditions hold:

i) $U_i \cap U_j = \emptyset$
ii) $U_j \cap Y_i = \emptyset$
iii) $Y_i \cap Y_j = \emptyset$ $\quad i,j \in \{ 1,...p\}$ where p is the total number of subroutines, n_i, n_j, m_i, $m_j \in N$ and U_i, Y_i, U_j, Y_j are the sets of input and output variables of subroutines i and j respectively.

3. A MATHEMATICAL MODEL OF SUBROUTINE DEPENDENCES

As defined before a subroutine can be viewed as a set of tasks which receives input variables and under some commands transforms these variables into output variables. This mapping can further be written as an explicit relation if an event-graph is used to describe the sequence of transformations which occur during the subroutine execution. These transformations will be called activities and their set will be denoted by A.. It has to be noted that a subroutine can also be viewed as an activity. The input and output variables are viewed as resources (R the set of all resources used in the program) for the subroutine execution. A program is then an acyclic oriented graph G which is assumed to be connected. The set of the arcs of the graph G is denoted by T. There is always a starting activity (node) a_s (r) and a final one a_f (r).

Each arc (i,j) $\in$ T of the graph G is weighted by an integer $t_{ij} \geq 1$ called the displacement. Each activity a_i will be executed following a certain path in the graph and consequently in an order

given by the precedence number x_i. If (j,i) G then $x_i=x_j+ t_{ij}$. A resource precedence number u_r will denote the moment the resource is used for the first time in the program. If i is the first activity for resource r, then $x_i \geq u_r$.

Consider now $P^-(i)$ the set of predecessors of activity i and $R^o(i)$ be the set of resources such that $a_S(r) = i$; then

$$a_i \in A \qquad x_i = \max\left(\max_{j \in P^-(i)}(x_j + t_{ij}), \max_{r \in R^o(i)} u_r\right) \qquad (1)$$

Let **A** be the nxn weighted incidence matrix of (A , T) defined by: $A_{ij} = t_{ij}$ if (i,j) T and $A_{ij}=-\infty$ otherwise, where n= Card(*A*). Similarly, let **B** be an rxn matrix, where r=Card (R) defined by $b_{ri}= 0$ if $a_S (r) =i$ and $b_{ri}= -\infty$ otherwise.

Using the above introduced matrices and the **minmax algebra** the following results can be obtained:

- The equation (1) can be written as

$$\mathbf{X = X A \otimes UB} \qquad (2)$$

where $X = (x_1,x_2, \ldots , x_n)$, $U= (u_1,u_2, \ldots , u_n)$

Letting yr denoting the precedence number of the activity where the resource is used for the last time, and $c_{ir}= t_i$ if $a_f(r)=i$ for some r and $c_{ir} =-\infty$ otherwise, a second relation is obtained:

$$\mathbf{Y= X C} \qquad (3)$$

- Theorem 1 : For a given U the equation $\mathbf{X = XA \otimes UB}$ has a unique solution : X= UBA* where $A^*= (E \otimes A \otimes A^2 \otimes A^{n-1})$, E is the identity matrix defined as $e^{ii}=0$ and $e_{ij}= -\infty$ for i=j, and $A^n=A^{n-1}$.

- Theorem 2:A* contains as entries the maximal weights of paths between two nodes and provides in this way the precedence numbers reflecting the activity dependences.
- Theorem 3 :If the critical graph of A* has only one path then there is a total dependence among the activities (subroutines) and their execution can only be sequential.
- Theorem 4 : if the critical graph of A* has K connected components, then there are K subroutines which can be executed in parallel.
- The previous results can be extended to the DO loop case. A DO loop can be seen as a part of a program which repeatedly performs the same activities over the same set of input and output variables. Using the index variable n X(n) will be the vector of the activities in the n-th run of the loop, and correspondingly U(n) will be the resource vector in the same run.This leads to the following model : U(n)=Y(n-1)K where K is an rxr matrix such that $K_{rs} =0$ for r=s and $K_{rr}=$ the displacement between $a_f(r)$ and $a_i(r)$. Using these observations and the previous results one can write

$$Y(n)=Y(n-1)\ KBA^*C \qquad (4)$$

which is a forward dynamic programming equation.

4. A LOGIC PROGRAMMING IMPLEMENTATION

The recurrence of the previous model suggests a logic programming implementation.

Using the model given by (1) & (3) , the results of theorem 1- 4 , and introducing the following recursive functions:

- find_calls(x,,first (l))= find_calls(x,,find_inner_sub(x, find_calls(x,first(new_l))))
- find_inner_sub(x,l)= find_inner_sub(x, find_calls(x,rest (l))

where first (l) and rest (l) are the head and the tail of the list l, and new_l is a working list containing at a certain moment the names of subroutines under processing

- norelation(): list(a_i(x_i,u_i), a_i(y_i)) list(s_i)
- $indx_i$ ($\cup_i$ a_i, $call_{ki}$) l_i, where l_i is a string of procedure names stored as a list

xi being defined by $indx_i()$={ l_{i1}, l_{i2} } and l_{i1} being the list of dependent subroutines,and l_{i2} being the list of independent ones.

With the above introduced recursive functions the main result can be stated as follows:

Proposition: For every list l of activities a_i $i \in N$, if the activities a_i are subroutines and if l is a nonempty list, the following recursive function

$indx_i$ (x , first(l))= x_i (x, rest(x_i (norelation (y_i,l_y),first(rest(l))))))

will build the list of all independent subroutines of the analyzed Fortran program.

An expert system shell has been used for the implementation of the above abstract mechanism for the detection of the subroutine dependences. An expert system has been obtained. The tasks accomplished by the expert system are : 1) the generation of an abstract Fortran file 2) the generation of input / output variable,and command lists, 3) the generation of all Fortran statements (ai) , 4) The generation of the corresponding xi,ui,yi,lists, 5) the dependency check 6) the generation of the lists of dependent and independent subroutines , 7) recursively repeats (6), 8) the displaying of the final lists.

The above implementation has been checked an examples of various degrees of difficulty. In a demonstration package a Fortran program with three levels of calls is considered for a dependency check. The expert system was implemented on a Vax and Compaq 386 environment.

5. CONCLUSIONS

The approach developed in this paper for detecting the subroutine dependences is based on a discrete event system model. The implementation has been accomplished by using a logic programming technique. To facilitate the implementation an expert system shell has been used. It provided the appropriate mechanism for reasoning, recreation, communication, and dynamic tracking of activities. It is a powerful and productive tool for developing software tasks previously implemented in compilers.

References:

1.R. Allen, K. Kennedy: Automatic Translation of FORTRAN Programs to Vector Form, ACM Transactions on Programming Languages and Systems, Vol. 9, NO. 4, Oct. 1987, pp 491-542.

2. C.D. Polychronopoulos, U.V. Banerjee:"Processor Allocation for Horizontal and Vertical Parallelism and Related Speedup Bounds", IEEE Transactions on Computers Vol. C-36, NO. 4, April 1987 pp 410-426.

3.C.D.Polychronopoulos,D.J.Kuck:"Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers", IEEE Transactions on Computers,Vol.c36,No.12, pp 1425-1439, Dec. 1987.

4. Kuck, D.J. The Structure of Computers and Computations, Vol 1, Wiley, New York, 1987.

5.Allew, J.R.: Dependence Analysis for Subscripted Variables and its application to Program Transformations, Ph.D. Dissertation, Dept of Mathematics & Computer Sciences, Rice University, Houston , Tx, April 1983.

6. P.Caspi, N. Halbwacks: "Functional Model for Describing and Reasoning about Time Behaviors of Computing Systems" , Acta Informatica ,Vol 22, pp 595-627, 1986.

7.R.A. Cuninghame Green: " Minimax Algebra" Lectures Notes in Economics and Mathematical Systems, vol. 166, New York, Springer-Verlag,1979.

8.P.W.Foulk, S.M. Nassar : " Analysis of Parallelism in Nested Do Loops" , The Journal of Systems and Software, no.5, 1985,pp73-80.

The CCS Interface Equation — an Example of Specification Construction using Rigorous Techniques

G Martin, R Everett
British Telecom Research Laboratories

April 3, 1989

Abstract

The use of algebraic techniques in the development of software enables systems to be built which have a precise formal foundation. Such formal methods[1] can help considerably in reducing the susceptibility to errors of interpretation and consistency of many of the current *ad hoc* procedures used in system design. This is of particular importance for building large and reliable systems. Furthermore, the use of formal techniques enables rigorous calculations of various properties to be made on a system like, for example, absence of deadlock.

In this paper we are concerned with how systems expressed in such a manner can be used to systematically generate specifications which may, in principle, be performed without any human intervention[2]. In particular, we deal with the type of situation where the specification of an unknown system component is derived from two given specifications. The pre-requisites for solution to the problem we consider are that the known specifications are formally expressed in terms of states and that the component to be synthesised interacts with them in some predefined way.

1 Notation

We are dealing with *finite communicating transition systems* as described in [3,4] in which α, β, γ, $\mu \ldots \overline{\alpha}$, $\overline{\beta}$, $\overline{\gamma}$ $\overline{\mu} \ldots$ denote *actions*, τ is a special action called the *invisible action* and p_1, p_2, q, $r \ldots$ are *states*. ϵ denotes an empty sequence of actions. A *direct* μ *derivation* between states is a relation between two states and and an action, written $p_1 \rightarrow^{\mu} p_2$, in which p_1 is the *source* and p_2 the *destination*. $R(p)$ is the set of all states reachable from p; $\Lambda(p)$ (called the alphabet of p) is the set of all actions which are possible for all $p' \in R(p)$. M_p is a machine which can exist in state p. For conciseness we write $p_1 \rightarrow^{\mu} p_2 \rightarrow^{\mu} p_3$ to mean $(p_1 \rightarrow^{\mu} p_2) \wedge (p_2 \rightarrow^{\mu} p_3)$. A set of relations sharing the same source is written as a *behaviour equation*: $p_1 \Leftarrow \alpha p_2 + \beta p_3$ means that $(p_1 \rightarrow^{\alpha} p_2) \wedge (p_1 \rightarrow^{\beta} p_3)$ and that there are no other relations having p_1 as a source. We express the ability of two transition systems to communicate by use of *complementary actions* such as μ and $\overline{\mu}$ and the parallel composition operator: $|$. If $p = \overline{\mu}p'$ and $r = \mu r'$ then $(p|r) = \overline{\mu}(p'|r) + \mu(p|r') + \tau(p'|r')$ ([3, Expansion Theorem]). That is, if p can do an action and r can do the complementary action, then the composition of the two can do either of these (in which case only one of the machines changes state) or it can perform the invisible τ action (in which case both machines change state). Given one machine we can derive another by 'hiding': that is, removing all transitions in which the action belongs to a set A; the τ action may not appear in such a set. In the previous example if we define a set $A = \{\mu\}$ then $(p|r)\backslash A = \tau(p'|r')$. By convention hiding an action implies hiding its complementary action. We define $p \Rightarrow^{\mu} p'$ to be a relation between two states p and p' and action μ ($\neq \tau$) if and only if

$$\underbrace{p \rightarrow^{\tau} p_1 \rightarrow^{\tau} p_2 \ldots \rightarrow^{\tau}}_{\text{0 or more } \tau\text{s}} \quad \underbrace{p_3 \rightarrow^{\mu} p_4}_{\text{exactly one } \mu} \quad \underbrace{\rightarrow^{\tau} p_5 \rightarrow^{\tau} \ldots \rightarrow^{\tau} p'}_{\text{0 or more } \tau\text{s}}$$

Replacing μ by τ in the above figure gives a representation of the definition of the relation $p \Rightarrow^{\epsilon} p'$ (where ϵ is the null string), which indicates that two states are connected by a sequence of zero or more τ actions. For conciseness we use $p \rightarrow^{\mu}$ (read as 'p can do a μ') to mean $\exists p'$ such that $p \rightarrow^{\mu} p'$ (but we are not interested in what p' is). *Observational equivalence* is defined in such a way that two states p and q are observationally equivalent, written $p \approx q$, if and only if for every *i)* μ if $p \Rightarrow^{\mu} p'$ then there exists a q' such that $q \Rightarrow^{\mu} q'$ and $p' \approx q'$ and *ii)* if $q \Rightarrow^{\mu} q'$ then there exists a p' such that $p \Rightarrow^{\mu} p'$ and $p' \approx q'$. *Weak determinacy* is defined in such a way that if $p \Rightarrow^{\mu} p'$ and $p \Rightarrow^{\mu} p''$ then $p' \approx p''$.

2 Interface Equation

An *interface equation* is an expression of the form $(p|X)\backslash A \approx q$ where $\Lambda(p) \cap \overline{\Lambda(q)} \subseteq \{\tau\}$, $\Lambda(p) \cap \overline{A} = \emptyset$ and $\Lambda(q) \cap (A \cup \overline{A}) = \emptyset$. ($\emptyset$ denotes the empty set.) We say that r is a solution to the equation $(p|X)\backslash A \approx q$ iff r satisfies $(p|r)\backslash A \approx q$ and $\Lambda(r) \cap \Lambda(p) \subseteq \{\tau\}$. In this paper we further assume that no pair of states q' and $q'' \in R(q)$ are observationally equivalent. This slightly simplifies the exposition and significantly reduces problems in implementing the algorithm. Since it is straightforward to compute from a machine M not having this property a new machine M' which has this property, the assumption does not impose significant constraints on the applicability of the theory which we develop.

The interface equation may be thought of as being approximately the reverse of the expansion theorem: whereas the expansion theorem composes two given machines to produce an unknown third we are attempting to compute the unknown machine which, when composed with a second, is observationally equivalent to a given third.

3 Methods of Solution

A basic procedure for solving the interface equation is a discarding algorithm very similar to that described in [5]. In this procedure we construct a set ψ, each component K of which is an *I-complete* (defined later) set of *tuples* of the form (p', q'), where p' and q' are states of the machines M_p and M_q. We then compute, for every pair (K, K') the relations (defined later) $K \rightarrow^{\mu,O} K'$, $K \rightarrow^{\mu,C} K'$ and $K \rightarrow^{\tau} K'$. We then scan through the components of ψ and discard any component K that is not *O-complete* (defined later) with respect to ψ. We iterate this scan until either no set remains that fails the O-completeness check (in which case we have found a solution) or none of the sets of ψ contains the tuple (p, q) (in which case no solution exists). The solution is expressed by creating one state r_i of the solution for each K in ψ. Derivations between r-states are readily associated with derivations between K-sets thus: if there is a $\rightarrow^{\tau}$ between two K sets there is a $\rightarrow^{\tau}$ between the corresponding r states; if there is a $\rightarrow^{\mu,O}$ or $\rightarrow^{\mu,C}$ between two K sets there is a $\rightarrow^{\mu}$ between the corresponding r states. The procedure of forming I-complete sets entails (in the most basic form of the algorithm) the formation of all possible valid unions of basic sets of tuples called $B_{I\tau}$ sets. Each of these $B_{I\tau}$ sets is I-complete, and the formation of valid unions consists of forming only those unions which retain this I-completeness condition.

4 Key steps in the theory

Relations between tuples: Let us first introduce relations $\rightarrow^{\mu,I}$ and $\rightarrow^{\tau,P}$ between tuples, where $\mu \in \Lambda(p) \cup \Lambda(q) - \{\tau\}$: define $(p', q') \rightarrow^{\mu,I} (p'', q'')$ iff $p' \rightarrow^{\mu} p''$ and $q' \Rightarrow^{\mu} q''$ and define $(p', q') \rightarrow^{\tau,P} (p'', q'')$ iff $p' \rightarrow^{\tau} p''$ and $q' \Rightarrow^{\epsilon} q''$. We then introduce sets of tuples $I_{\mu,I}$ and $I_{\tau,P}$: define $I_{\mu,I}(p', q', p'') = \{(p'', q'') | (p', q') \rightarrow^{\mu,I} (p'', q'')\}$ and $I_{\tau,P}(p', q', p'') = \{(p'', q'') | (p', q') \rightarrow^{\tau,P} (p'', q'')\}$. We also need two further relations between tuples $\rightarrow^{\mu,O}$ and $\rightarrow^{\mu,C}$: define $(p', q') \rightarrow^{\mu,O} (p'', q'')$ iff $p' = p''$ and $q' \Rightarrow^{\mu} q''$ and $\mu \in \Lambda(q) - \Lambda(p)$ and $\mu \neq \tau$; define $(p', q') \rightarrow^{\mu,C} (p'', q'')$ iff $p' \rightarrow^{\overline{\mu}} p''$ and $q' \Rightarrow^{\epsilon} q''$ (where ϵ is the null string) and $\mu \in A$ and $\mu \neq \tau$.

$B_{I\tau}$ sets: A key step in developing efficient algorithms is the introduction of $B_{I\tau}$ sets, which are defined in terms of the preceding sets, as follows. Define $B^{(i)}_{I\tau}(p', q')$ ($i \in \{1, 2 \ldots\}$) by the following: $(p', q') \in B^{(i)}_{I\tau}(p', q')$; if $(p'', q'') \in B^{(i)}_{I\tau}(p', q')$ and $(I_{\mu,I}(p'', q'', p''') \neq \emptyset)$ or $(I_{\tau,P}(p'', q'', p''') \neq \emptyset)$ then (p'', q'') is any one of the tuples in $I_{\mu,I} \cap I_{\tau,P}$ (if both are non-empty) or any tuple in the non-empty set if only one of them is empty. If q is not weakly determinate then for a given (p', q') there may be several $B_{I\tau}$ sets depending on which of the tuples is selected; the superscript (i) is used to distinguish between these sets.

I-completeness: I-completeness is a property of a set K of tuples, K being a union of $B_{I\tau}$ sets. Such a set is I-complete iff $\forall (p', q') \in K$, if $p' \rightarrow (\mu \notin A$ and $\mu \neq \tau)$ then $q' \Rightarrow^{\mu}$.

5 Relations between sets

We define five relations between sets: $\rightarrow^{\mu,O}$, $\rightarrow^{\mu,C}$, $\rightarrow^{\tau}$, $\Rightarrow^{\mu,O}$ and $\Rightarrow^{\mu,C}$. Define $K \rightarrow^{\mu,O} K''$ iff $\forall (p', q') \in K \exists (p'', q'') \in K'$ *s.t.* $(p', q') \rightarrow^{\mu,O} (p'', q'')$. Define $K \rightarrow^{\mu,C} K'$ iff $((\exists (p', q') \in K)$ and $(\exists (p'', q'') \in K'')$ *s.t.* $(p', q') \rightarrow^{\mu,C} (p'', q''))$ and $(\forall (p', q') \in K$ if $p' \rightarrow^{\overline{\mu}} p''$ with $\overline{\mu} \in A$ then $\exists (p'', q'') \in K'$ *s.t.* $(p', q') \rightarrow^{\mu,C} (p'', q''))$. Define $K \Rightarrow^{\mu,O} K''$ iff there exists a sequence of K such that

$$\underbrace{K \rightarrow^{\tau} K_1 \rightarrow^{\tau} K_2 \cdots \rightarrow^{\tau} K_3}_{\text{zero or more } \tau s} \quad \underbrace{\Rightarrow^{\mu,O}}_{\text{exactly one } \mu, O} \quad \underbrace{K_4 \rightarrow^{\tau} K_5 \cdots \rightarrow^{\tau} K'}_{\text{zero or more } \tau s}$$

Similar definitions are made for μ, C and τ derivations.

O-completeness A set K is said to be O-complete with respect to a set S if for all $(p', q') \in K$ if $\mu \in \Lambda(q)$ and $q' \rightarrow^{\mu} q''$ then there exists a sequence of sets K_i and tuples (p_i, q_i) (where each $(p_i, q_i) \in K_i$) such that

$$\begin{array}{l} p' \Rightarrow^{\mu_1} p_1 \Rightarrow^{\mu_2} p_2 \Rightarrow^{\mu_3} \cdots p_4 \Rightarrow^{X} p_5 \Rightarrow^{\mu_6} \cdots \Rightarrow^{\mu_7} p' \\ K \Rightarrow^{\overline{\mu}_1, C} K_1 \Rightarrow^{\overline{\mu}_2, C} K_2 \Rightarrow^{\overline{\mu}_3, C} \ldots K_4 \Rightarrow^{Y} K_5 \Rightarrow^{\overline{\mu}_6, C} \ldots \Rightarrow^{\overline{\mu}_7} K' \end{array}$$

and $(p', q'') \in K'$ where, if $\mu = \tau$ then $X = \epsilon$ and $Y = \epsilon$, otherwise *either* $X = \mu$ and $Y = \epsilon$ *or* $X = \epsilon$ and $Y = \mu, O$ Notice that I-completeness of a set is not influenced by other sets; in contrast, O-completeness, while still being a property of a set, is influenced by that set's relations with other sets.

6 Key steps in algorithm development

The practical problem of this basic approach its combinatorial complexity, which causes the number of sets in ψ to be very large. This complexity arises (*i*) the formation of all possible valid unions of $B_{I\tau}$ sets, and (*ii*) the relaxation of weak determinacy, which considerably increases the number of $B_{I\tau}$ sets[6].

We have attempted to reduce the computational requirements in two ways. First we have refined the basic procedure described above by introducing the concept of *minimal unions*[7]. This concept considerably reduces the number of sets that we have to deal with in performing the O-completeness tests. Secondly, we have attempted a constructive approach[8] instead of a discarding approach.

The minimal-union approach to improving the basic discarding algorithm is to try to avoid forming unions which are not essential to a solution. We can do this by first considering *images* of derivations between sets. For example, the image of a τ derivation from K to K' is the set $L' \subseteq K'$ given by $\{(p'', q'') | (p'', q'') \in K' \wedge (\exists (p'', q') \in K \; s.t. \; q' \Rightarrow^{\epsilon}$ where ϵ is the null string)$\}$. Similar definitions are made for μ, X and μ, O derivations. A minimal union from K containing K' is defined to be the union of only those $B_{I\tau}(p', q')$ sets with $(p', q') \in K'$ such that $B_{I\tau}(p', q') \subseteq K$. We have found that it is sufficient to consider only minimal unions of images of μ, C, μ, O and τ derivations in computing solutions, allowing a considerable reduction in computing requirements in some examples.

In the constructive approach we delay the steps which generate the large number of sets as long as possible. Instead of performing the single linear sequence of forming $B_{I\tau}$ sets, I complete sets, unions, and then carrying out the O-completeness tests, we carry out an iterative procedure. In this iterative procedure we do not immediately set up I-complete sets but set up what we call $\hat{I}$-complete sets (which at any stage we can expand to produce I-complete sets). We then test these sets for a more complicated for of O-completeness which we term $\hat{O}$-completeness. If this test fails on some set K, we then derive from the offending set more $\hat{I}$-complete sets by a heuristic process outlined in the next section. In this way we hope to avoid the exponential explosion of states when attempting to solve real problems.

7 Constructive algorithm concepts

In the constructive algorithm as well as eliminating unnecessary generation of $B_{I\tau}$ sets, or at least postponing such generation to a late stage, we pre-process the q machine into a *minimum action representation*. This produces a new machine observationally equivalent to the original q but with new and useful properties. The minimum action representation of a machine M is derived from the original machine by removing all derivations $q \rightarrow^{\mu} q'$ which are not observationally essential. A derivation of the form $q \rightarrow^{\mu} q'$ (where $\mu \in \Lambda(q) - \{\tau\}$)is observationally essential iff $\forall q'' \; s.t. \; q'' \not\approx q$ then $q \Rightarrow^{\epsilon} q'' \Rightarrow^{\mu} q'$ is false and $\forall q'' \; s.t. \; q' \not\approx q''$ then $q \Rightarrow^{\mu} q'' \Rightarrow^{\epsilon}$ is false. A similar definition is made for derivations of the form $q \rightarrow^{\tau} q'$. Analagous definitions about the minimum paths between tuples can be made. By analysing the reachability of the p machine by actions not in A and by comparing minimum paths of the p machine with minimum paths between (p, q) tuples, we compute a relation $\mathcal{R}_S$ between tuples. (The lengthy definitions of $\mathcal{R}_S$, $\hat{I}$-completeness and $\hat{O}$-completeness have been omitted here.) We then construct the transitive closure of $\mathcal{R}_S$ and examine the set of equivalence classes given by this relation. By construction each of these sets satisfies our definition of $\hat{I}$-completeness. We then check each of these sets for $\hat{O}$-completeness. In contrast to the discarding algorithm, we do not discard a set which fails. Instead, we replace the offending set K by two sets, one derived from K by deletion of some tuple (p', q') and the other derived from K by retention of (p', q') and deletion of all other tuples beginning with p'. This process is known as tuple extraction. In general the resulting sets are not $\hat{I}$-complete, and so we have to refine the sets (by tuple deletion) until they are $\hat{I}$-complete. Deletion of tuples also implies that the $\mathcal{R}_S$ relations have to be recomputed. We

iterate round the cycle of $\hat{I}$-completness checking, $\hat{O}$-completeness checking and tuple extraction, until the $\hat{O}$-completeness condition is satisfied.

The process of tuple extraction is a heuristic one. That is, we have worked out rules for guessing which choice of tuple is most likely to lead quickly to a solution. The algorithm we use would eventually search through all tuples, though this would take far too long in general. Consequently appropriate selection of tuples is of crucial importance in this step.

8 Computation times

Using a discarding algorithm, an M_p with two states and a M_q with three states can be solved in a few seconds on a VAX785. An example M_p with 5 states and an M_q with 20 states but requiring no unions was on the limits of solubility (1 CPU-day). We have not yet explored the computation time of the constructive algorithm, but we expect the 5/20 example to be soluble in a few minutes of CPU time.

9 Conclusions and current work

We have produced algorithms which solve the interface equation for q machines which are not weakly determinate. Basic versions of these algorithms require impracticably large computation time for machines with more than a very small number of states. More advanced versions are under development which are more likely to produce solutions in a reasonable amount of time. Current work involves the optimisation of the heuristics for tuple selection in the constructive algorithm.

Acknowledgements: The authors would like to thank the director of British Telecom Research Laboratories for permission to publish this paper. Thanks are also due to C Osborn and B Lloyd (both of the System and Software Engineering Division) for translating abstract mathematics into working Pascal programs.

References

[1] B Cohen. Justification of formal methods for system specification. *Software and Microsystems*, 1(5), 1982.

[2] M T Norris, R P Everett, G A R Martin, and M W Shields. Method for the synthesis of interactive system specifications. *J. Info. and Software Tech.*, 30(7), September 1988.

[3] R Milner. *A Calculus of Communicating Systems.* Volume 92 of *Lecture Notes in Computer Science*, Springer-Verlag, 1980. ISBN 3-540-10235-3.

[4] M W Shields. *An Introduction to Automata Theory.* Blackwell Scientific Publications, 1987. ISBN 0-632-01554-3.

[5] M W Shields. *Solving the Interface Equation, Tech. Rep. SE/079/2.* Technical Report, Electronic Engineering Laboratories, University of Kent at Canterbury, July 1986.

[6] G Martin. *A General Solution to the Interface Equation.* Technical Report, British Telecom Research and Technology Internal Memorandum Number R11/87/023, 1987.

[7] G Martin. *Practical Considerations of Applying the Interface Equation.* Technical Report, British Telecom Research and Technology Internal Memorandum Number R11/87/007, 1987.

[8] G Martin. *A Study into a Constructive Method for Generating Solutions to the Interface Equation.* Technical Report, British Telecom Research and Technology Internal Memorandum Number RT31/89/004, 1989.

Implementing Mathematics as an Approach for Formal Reasoning

Robert L. Constable
Department of Computer Science
306 Upson Hall
Cornell University
Ithaca, NY 14850

Two simple but important algorithms used to support automated reasoning are tautology checking and matching. Given two terms matching produces a substitution, if one exists, that maps the first term to the second. In this lecture these two algorithms are used to illustrate the approach to automating reasoning suggested in the title. Both algorithms can be derived and verified in the Nuprl proof development system following exactly the informal presentation we use here.

These examples serve to introduce a particular automated reasoning system, Nuprl, as well as the idea of deriving programs from constructive proofs. The treatment of the examples also suggests how these systems can be soundly extended by the addition of constructive metatheorems about themselves to their libraries of results.

Pairings on Lambda Algebras

W. S. Hatcher, Université Laval, Canada and
Marcel Tonga, Université d'Yaoundé, Cameroun

This paper continues the authors' universal algebraic approach to the study of the λ-calculus begun in [Hatcher & Tonga 1985] and further developed in [Hatcher & Scott, 1986] and [Tonga 1987]. The basic insight underlying this approach is that the traditional λ-calculus is algebraically defective because it uses only one-half of the natural isomorphism $A^{B\times C} \cong (A^C)^B$, namely the right-hand side. The principal means of removing this defect is by an appropriate theory of pairings on so-called λ-algebras.

Let $\mathcal{A} = (A,')$ be a groupoid (' is a binary operation, called application, on the non-empty set A). We assume $|A| > 1$ throughout. $\mathcal{A}$ is a λ-system if it supports a syntactically appropriate λ-operator satisfying the conversion identities $(\lambda x t)'x \equiv t$, where t is any A-term, i.e., a term of the first-order diagram language L(A) of groupoids over A (thus an element of the underlying set $W_A(X)$ of the absolutely free (word-) algebra $\mathcal{W}$ of groupoids with distinguished constants A and variables X) and ≡ is the minimal congruence relation on $W_A(X)$, obtained by taking all possible evaluations of X in A). Thus, a λ-system $\mathcal{A}$ has constants $K, S \in A$ such that K'a'b = a and S'a'b'c = (a'c)'(b'c) hold, where a, b, and c are any elements of A (parenthesis-free iterations of application are associated to the left). A λ-system satisfying all universal A-λ-identities (see [Hatcher & Scott 1986]) is a λ-algebra, and a λη-algebra if it satisfies the further identity (η): $\lambda x(t'x) \equiv t$, where t is any A-term.

A pairing is defined on a nonempty set A whenever A supports a binary coupling operation [a,b] and unary projection operations p(x) and q(x) satisfying the identities p([a,b]) = a and q([a,b]) = b. If the set A is the support of a λ-system $\mathcal{A}$, then a pairing with binary coupling [a,b]

is defined on A precisely when there exist nullary operations π_1 and π_2 on A satisfying the identities $\pi_i'[a_1,a_2] = a_i$: given π_1 and π_2, $p(x) = \pi_1'x$ and $q(x) = \pi_2'x$, while, conversely, $\pi_1 = \lambda x p(x)$ and $\pi_2 = \lambda x q(x)$ when $p(x)$ and $q(x)$ are given.

A pairing is usually defined on λ-systems by: $[a,b]^\lambda = \lambda x(x'a'b)$; $\pi_1^\lambda = \lambda x(x'K)$; $\pi_2^\lambda = \lambda x(x'K'(S'K'K))$ (see e.g. [Barendregt 1984]). However, this pairing has undesirable special properties. For example, it satisfies the condition of surjectivity, $[\pi_1^\lambda'a,\pi_2^\lambda'a]^\lambda = a$, only when $|A| = 1$ (see [Tonga 1987, p. 18]). We call this pairing canonical to distinguish it from others.

This difficulty concerning the canonical pairing is overcome by extending the language L(A) of groupoids to a language $L_\pi(A)$ that includes new constants π_1 and π_2 and a second binary operation $[-,-]$. The structure $\mathcal{A} = (A, ', [-,-], \pi_1, \pi_2)$ is a coupled groupoid when it satisfies the identities $\pi_1'[a,b] = a$ and $\pi_2'[a,b] = b$ for all $a, b \in A$. A coupled groupoid is a λ-π-system if it supports a syntactically appropriate λ-operator satisfying conversion identities for all terms t of $L_\pi(A)$. We have:

Theorem 1. A coupled groupoid $\mathcal{A}$ is a λ-π-system if and only if A has constants S, K, and W satisfying the identities $K'a'b = a$; $S'a'b'c = (a'c)'(b'c)$; $W'a'b'c = [a'c, b'c]$, where a, b, and c are any elements of A. ■

This combinatory form of λ-π-systems is very helpful in studying the relationship between various pairings defined on them. Indeed, we can use the λ-operator in a λ-π-system to define pairings other than the one given by the primitive coupling operation $[-,-]$ and the primitive constants π_1 and π_2. An important and useful example is the following:

A λ-π-system satisfying all universal A-λ-π-identities (see [Tonga 1987, p. 23]) is a λ-π-algebra, and a λ-π-algebra satisfying the identity (η) is a λ-η-π-algebra. Let $\mathcal{A}$ be a λ-η-π-algebra. Then, $\langle a,b\rangle = \lambda x[a'x,b'x]$, $p(x) = \lambda y(\pi_1'(x'y))$, and $q(x) = \lambda y(\pi_2'(x'y))$ define a pairing on A, called a function pairing. It is a pairing frequently used when dealing with the "monoid form" of a λ-η-π-system, in which the λ-operator is used to define the following further operations. (1) Composition: $a \circ b = \lambda x(a'(b'x))$. (2) Identity: $I = \lambda x(x)$. (3) Exponentiation

(Currying up): $g^* = \lambda x(\lambda y(g'[x,y]))$. (4) Extraction (Currying down): $\sqrt{g} = \lambda x(g'(\pi_1'x)'(\pi_2'x))$. (5) Evaluation: $\varepsilon = \sqrt{1}$. If the original pairing is the canonical one, then the derived function pairing is called <u>standard</u>. The standard function pairing can be given an intrinsic definition in terms of the monoid structure of the system (see [Tonga 1987, p. 60]).

Suppose, now, that we are given a monoid $(M, \circ, 1)$ enriched with a further binary operation $\langle -, - \rangle$, a unary operation *, and nullary operations π_1, π_2, and ε. Then $\mathcal{M} = (M, \circ, 1, \langle -, - \rangle, , ^*, \pi_1, \pi_2, \varepsilon)$ is a <u>weak C-monoid</u> if these data satisfy the following identities: $\pi_i \circ \langle a_1, a_2 \rangle = a_i$; $\langle a, b \rangle \circ c = \langle a \circ c, b \circ c \rangle$; $\varepsilon \circ \langle a^* \circ \pi_1, \pi_2 \rangle = a \circ \langle \pi_1, \pi_2 \rangle$; $a^* \circ b = (a \circ \langle b \circ \pi_1, \pi_2 \rangle)^*$.

<u>Theorem 2</u>. Any weak C-monoid is, under the appropriate definitions, a λ-π-algebra. Conversely, any λ-η-π-algebra is, under the definitions given above, a weak C-monoid. ■

Only the ($\Rightarrow$) half of Theorem 2 is really new (although the converse was in fact established only for the standard pairing, see [Adachi 1983] and [Koymans 1984]). The proof uses a variant, due to Tonga, of the discriminant of [Hatcher & Scott 1986].

A weak C-monoid $\mathcal{M}$ is a <u>λ-monoid</u> if the identity $\varepsilon = \varepsilon \circ \langle \pi_1, \pi_2 \rangle$ holds in $\mathcal{M}$, an <u>η-monoid</u> if the identity $\varepsilon^* = 1$ holds, and a <u>surjective monoid</u> if $\langle \pi_1, \pi_2 \rangle = 1$. Finally, a surjective η-monoid is a <u>C-monoid</u>.

<u>Theorem 3</u>. Any λ-η-monoid is a λ-η-π-algebra and, conversely, any λ-η-π-algebra is a λ-η-monoid. ■

This result is contained in [Tonga 1987]. It strictly generalizes the main result of [Hatcher & Scott 1986], employing similar techniques and using the result of Theorem 2 above as a lemma.

In fact, Theorem 3 is a (particularly useful) special case of the following theorem:

<u>Theorem 4</u>. There is an equivalence of categories between the category of all λ-π-algebras and the category of all λ-monoids. ■

The special case of Theorem 4 obtained by taking only the canonical pairing for the λ-algebras and the standard pairing for λ-monoids is the well-known result of [Adachi 1983] and [Koymans 1984].

Finally, we give necessary and sufficient conditions for any two pairings to be pairings for the same, given λ-monoid structure. This generalizes a similar result for C-monoids found in [Lambek & Scott 1986].

The results of the present paper show that most of the various structures which serve as models for the λ-calculus have, when endowed with a pairing system, elegant algebraic formulations as monoids. In particular, we have obtained these results without ever imposing the condition of surjectivity on our pairings. Yet, all of the results extend easily to the surjective case.

Important for computer science and the theory of recursive functions in general is the fact that none of the various structures dealt with in this study are required to be extensional (or even weakly extensional).

References

T. Adachi [1983]: A categorical characterization of lambda calculus models, Research Report on Information Sciences, Tokyo Institute of Technology, No. C-49.

H. P. Barendregt [1984]: The Lambda Calculus, Revised Edition, North-Holland, Amsterdam.

W. S. Hatcher & P. J. Scott [1986]: Lambda-algebras and C-monoids, Zeit. Math. Log. Grund. Math., Vol. 32, pp. 415-430.

W. S. Hatcher & M. Tonga [1985]: Equational extensions of classical combinatory logic, Abstracts, Amer. Math. Soc., Vol. 6, pp. 226-227.

J. Lambek & P. J. Scott [1986]: Introduction to Higher-Order Categorical Logic, Cambridge University Press, Cambridge, England.

C. P. J. Koymans [1984], Models of The Lambda Calculus, Thesis, Rijksuniversiteit, Utrecht, Holland.

M. Tonga [1987]: Couplage sur les lambda-algèbres, Thesis, Université Laval, Québec, Canada.

Constructor Model as Abstract Data Types
(summary)

Hantao Zhang
Department of Computer Science
The University of Iowa
Iowa City, IA 52242
hzhang@herky.cs.uiowa.edu

Data abstraction has been widely recognized as an important technique for designing programs and the notion of *Abstract Data Types* (ADT for short) was invented for formally studying these abstraction techniques. In particular, *equational specifications* of ADT, which posit a set of many-sorted algebraic objects to a finite set of equations, enjoy considerable popularity because programmers can easily formalize equations within programming languages while pure mathematicians can easily study the algebraic objects specified by equations. However, many people feel that this approach creates more problems than it solves [ManesArbib 86,Ch.14]. Such kind of frustration comes, we believe, partially from definitions of ADT or equivalently, interpretations of equational specifications.

According to the ADJ group [Goguen et al 75], an ADT is an isomorphic class of universal algebras. They proposed that the class of initial models (unique under isomorphism), which are the minimal algebraic structure satisfying the given equations, are used as the interpretation or semantics of specifications [Goguen et al 75]. In this initial-model approach, all functions in a specification are considered uniformly.

Another interesting isomorphic class of universal algebras is the ones isomorphic to the set of algebraic objects built up uniquely by *constructors*. In this approach, all functions in a specification are explicitly classified into constructors and nonconstructors (or destructors), with the interpretation that the constructors and equations on constructors define the model (called *constructor model*) of a specification. This approach to equational specifications is not new: Peano's arithmetics, Boyer and Moore's shell-principle, etc., can be considered as instances of this approach.

A serious problem in the initial-model approach is to handle erroneous and meaningless expressions as well as incompletely defined nonconstructors. When some incomplete functions are presented, it is often difficult to reconcile the initial object condition with intuitively correct equations of the intended model. For example, suppose that a specification defines '+' over natural numbers with the equations $E = \{\ 0+x=x,\quad suc(x)+y=suc(x+y)\ \}$. In this case, the initial model of E is (isomorphic to) the natural number set and the equation $x+y=y+x$ is true in the initial model. If E is extended by adding a single equation $pre(suc(x)) = x$, then $x+y=y+x$ is no longer true in the new initial model, since the function pre is not defined on 0. If $pre(0)$ is substituted for x and 0 for y in $x+y=y+x$, the resulting two sides are not congruent. This is not very surprising as the initial object set can be changed with the addition of a new function symbol which may introduce new values. The side effect of this change is that the new initial model is often very hard to describe and is no longer the

intended model.

To overcome this problem, many attempts have been tried. The *sufficient completeness* property of ADT specifications introduced by Guttag [Guttag 75] has been found useful. A specification is sufficiently complete if every nonconstructor is completely defined over constructors. However, requiring every specification being sufficiently complete is often inconvenient and is too restrictive in a system for building specifications.

It is always possible that the intended model can be built up by a minimal set of operators (called *constructors*). As long as the constructor set and their relations are fixed in a specification, we may consider that the model of the specification remains unchanged, no matter what functions have been added and whether these new functions are completely defined. This intended model is what we call "constructor model".

For a given specification, the constructor model has a close relation with the initial model. In terms of universal algebras, the constructor model is just a subalgebra of the initial model with constructor terms as its domains.

Definition 1 (subalgebra) *Given a signature (S, F) and an F-algebra $A = (S_A, F_A)$, where S_A is the domain of A and F_A, the functions of A. An F-algebra $B = (S_B, F_B)$ is said to be a subalgebra of A if* (i) $F_B = F_A$ *and* (ii) *for each $A_s \in S_A$ and $B_s \in S_B$, we have $B_s \subseteq A_s$, where $s \in S$, A_s and B_s are the object set of sort s in A and B, respectively.*

In contrast to the classical definition [BirkhoffLipson 70], we do not require that the domains of a subalgebra be closed under its operations. This is because all the functions are total in an initial algebra. If we had required that subalgebras be closed under functional application, then an initial algebra could not have any non-trivial subalgebras, except for the case where some domains of such subalgebras are void.

Let $I(F, E)$ denote the initial algebra specified by a signature (S, F) and a set E of equations. We are interested in subalgebras of $I(F, E)$ such that the domains of such a subalgebra are not void for each sort $s \in S$ and are determined by specifying a subset of F. More precisely, for a subset $F' \subseteq F$, we require the domains of its subalgebra to be isomorphic to the free term algebra $T(F')$ modulo the congruence $=_E$. We say that they are subalgebras of $I(F, E)$ with respect to F' and write $I(F/F', E)$ to denote them.

Theorem 2 *Given S, F, E and $F' \subseteq F$. The following statements are equivalent:*

(a) *$I(F, E)$ is isomorphic to $I(F/F', E)$;*

(b) *The functions of $I(F/F', E)$ are total;*

(c) *The $E-$congruence classes (modulo $=_E$) of the term algebras $T(F)$ and $T(F')$ are isomorphic.*

Given a signature (S, F), a subset C of F and a set E of equations over F and variables, let us denote an equational specification by $SP = (S, F, C, E)$, where C is called the *constructors* of SP and $F - C$, the nonconstructors of SP.

Definition 3 (constructor model) *Given a specification $SP = (S, F, C, E)$, the* constructor model *of SP is the subalgebra $I(F/C, E)$ of the initial model $I(F, E)$ with respect to C.*

Example 4 Let $SP = (S, F, C, E) = (\{iowa\}, \{0, suc, pre\}, \{0, suc\}, \{pre(suc(x)) = x\})$. The domain of sort *iowa* in the initial algebra of SP is neither the natural number set nor the integer set, it can be represented by:

$$\{suc^i(pre^j(0)) \mid i,j \in I\!N, \text{the natural number set}\},$$

which is isomorphic to $I\!N \times I\!N$. An initial algebra of SP is

$$I(F,E) = (I\!N \times I\!N, \{0_I, suc_I, pre_I\})$$

where

$$\begin{array}{lcl} 0_I & = & \langle 0,0\rangle, \\ suc_I & = & \lambda\langle i,j\rangle.\langle i+1,j\rangle, \\ pre_I & = & \lambda\langle i,j\rangle.\textbf{if } (i=0) \textbf{ then } \langle i,j+1\rangle \textbf{ else } \langle i-1,j\rangle. \end{array}$$

The subalgebra of the initial model with respect to $C = \{0, suc\}$ is:

$$I(F/C,E) = (I\!N \times \{0\}, \{0_I, suc_I, pre_I\})$$

where 0_I, suc_I and pre_I are the same as in $I(F,E)$ above. By definition, $I(F/C,E)$ is the constructor model of SP above. Note that the function pre_I is not total in $I(F/C,E)$, since $pre_I(\langle 0,0\rangle) = \langle 0,1\rangle$, which does not belong to the domain of $I(F/C,E)$.

By definition, the subalgebra of the initial model with respect to constructors is the constructor model of an equational specification. It is easy to derive from Theorem 2 that if a specification is sufficiently complete, then the initial model and the constructor model of a specification are isomorphic. In general, the constructor model and the initial model are different. The advantage of the former over the later is that it is easy to reconcile the initial object condition among constructor terms with the intended model and to reconcile the soundness of equations in the constructor model with intuitively correct equations. Hence, it is more natural and intuitive to use constructor model as the ADT of a specification.

Constructor model can be also considered as the initial model of the sub-specification of a specification obtained by ignoring any nonconstructors. In other words, every constructor model has an *initial model specification*. Because of the initiality of constructor models, the constructor model approach inherits almost any advantage of the initial model approach for ADT specifications. Like initial models, constructor models can be used to justify the correctness of equational programs. Because every value can be represented by a syntactical term, a computing step corresponds to a deduction step from a term t_1 to another term t_2 by a set of inference rules. The correctness of the computing is thus reduced to the validity of the equation $t_1 = t_2$ in the constructor model. Similarly, the validity of inference rules can be also verified in the constructor model. If the equations in a specification possesses a canonical rewrite system in which if the right side of a rewrite rule has non-constructors, then its left side has also non-constructors (called *constructor preserving* [Kapur et al 85]), then the constructor model is computable because the domain of the constructor model is the same as the collection of all the normal forms of ground constructor terms.

In [Zhang 88], the constructor model has been used to establish the soundness of inductive theorem proving techniques and to characterize different classes of theorems of an equational specification. It is shown that the class of all the equations valid in the constructor model is a superset of the equations valid in the initial model, but is a subset of the equations which can be proved if we add one more inference rule called *full consistency* to the proof system (see also [KapurMusser 84]). It is also shown that the induction principle based the constructor model, with the rules of equational reasoning together, constitutes a set of inference rules that

has the monotonicity property with extension, a desirable property for automated reasoning systems.

In [ManesArbib 86](pp.327), it is criticized that the initial model approach does not provide a satisfactory explanation on the relation between two ADT *stack* and *queue*. However, it becomes clear when we compare their constructor sets because they have the same constructors (after renaming). Both of them can be implemented by the ADT *list* because *list* not only has the same constructor set as that of *stack* and *queue*, but also has a richer set of nonconstructors than *stack* and *queue*.

Finally, it is worth mentioning that the order-sorted algebra approach by [Goguen et al 85] is compatible with the constructor-model approach and their results (including the implementation results in OBJ3 [GoguenWinkler 88]) can be carried over in a natural form.

Acknowledgement: Thanks to Monagur Muralidharan for his useful comments on an earlier draft of this note.

References

[BirkhoffLipson 70] Birkhoff, G., Lipson, J., Heterogeneous algebras, *Journal of Combinatorial Theory*, 8, 1970. 115-113.

[Goguen et al 75] Goguen, J.A., Thatcher, J.W., Wagner, E and Wright, J.B., "Abstract data types as initial algebras and the correctness of data representations", In *Computer Graphics, Pattern Recognition and Data Structure*, IEEE, 1975, 89-93.

[Goguen et al 85] Goguen, J.A., Jouannaud, J.P., and Meseguer, J., "Operational semantics of order-sorted algebra". In *Proc. of Inter. Conference on Automata, Languages and Programming*, Brauer, W., Ed. LICS 194, Springer-Verlag, 1985.

[GoguenWinkler 88] Goguen, J.A., and Winkler, T., Introducing OBJ3, SRI-CSL-88-9, Computer Science Laboratory, SRI International, August 1988.

[Guttag 75] Guttag, J., *The Specification and Application to Programming of Abstract Data Types.* Department of Computer Science, Univ. of Toronto, Ph.D. Thesis, CSRG-59, 1975.

[KapurMusser 84] Kapur, D., and Musser, D.R., "Proof by Consistency," *Proc. of an NSF Workshop on the Rewrite Rule Laboratory*, Sept. 4-6, 1983. Schenectady, G.E. RD Center Report GEN84008, April 1984. (also in *Artificial Intelligence* 31, 1987, 125-57).

[Kapur et al 85] Kapur, D., Narendran, P., and Zhang, H., "On sufficient completeness and related properties of term rewriting systems". GE Corporate Research and Development Report, Schenectady, NY. Also in *Acta Informatica*, Vol. 24, Fasc. 4, August 1987, 395-416.

[ManesArbib 86] Manes, E.G., and Arbib, M.A., Algebraic Approaches to Program Semantics, Springer-Verlag, 1986.

[Zhang 88] Zhang, H. *Reduction, Superposition and Induction: Automated Reasoning in an Equational Logic*, Ph.D. Thesis, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, August 1988. Also in: Technical Report 88-06, Dept. of Computer Science, Univ. of Iowa.

LCF Should Be Lifted (Summary)

Bard Bloom* Jon G. Riecke*
MIT Laboratory for Computer Science
Cambridge, MA 02139
email: bard@theory.lcs.mit.edu, riecke@theory.lcs.mit.edu

Abstract: When observing termination of closed terms at all types in Plotkin's interpreter for PCF [11], the standard cpo model $\mathcal{A}_{\vee}$ is not adequate. We define a new model, $\mathcal{A}_{\vee}$, with *lifted* functional types and prove its adequacy for this notion of observation. We prove that with the addition of a parallel conditional and a convergence testing operator to the language, the model becomes fully abstract; with the addition of an existential-like operator, the language becomes universal. Using the model as a guide, we develop a sound logic for the language.

1 Introduction

The denotational semantics most appropriate for a programming language depends crucially upon the observations one makes about computations. In general, an **observation** is some important behavior of the interpreter [8]. For example, in the arithmetic, higher-order programming language PCF [11, 13], one usually chooses to observe the results of arithmetic expressions—that a term of integer type reduces to a numeral. One may also extend the notion of observation to arbitrary terms, saying that two terms are **observationally congruent** if they produce the same observable outcomes in any program context.

A good denotational semantics should be able to predict the observational behavior of a term. Each observation must therefore have a denotational meaning. When observing numerals in PCF, for example, M evaluates to 78 should imply that M means 78. If the converse holds as well, we say that the semantics is adequate. A perfect match occurs when observational congruence and semantic equality coincide; the semantics is then called **fully abstract.**

The language PCF, when observing numerals, has a well-matched denotational semantics. Plotkin and Sazonov show that the Scott-style, cpo model $\mathcal{A}_{\vee}$ is adequate [11, 12]. Moreover, although $\mathcal{A}_{\vee}$ is *not* fully abstract, the addition of a parallel conditional operator pcond to PCF makes the model fully abstract under this notion of observation [11, 12].

There may be other plausible choices for observations, *e.g.*, in a language with stores, one could observe the contents of memory cells. Other notions of observation can open a morass of problems. In PCF, for example, one might wish to observe terms at higher type, *e.g.*, printing a message when a term "equals" the identity function $\lambda x.x$. One must then choose the sense in which to compare terms of functional type: syntactic equality is probably too fine-grained, whereas observational congruence of terms is undecidable [17]. In particular, we cannot hope to observe the identity function in the same way we do numerals.

Nevertheless, one may reasonably observe *termination* of terms of functional type. When given a term of higher type, Plotkin's interpreter for PCF will either terminate at a λ-abstraction or diverge. For example, let Ω^{σ} be a term of type σ that diverges, and consider the two PCF terms $\lambda x^{\tau}.\Omega^{\tau}$ and $\Omega^{\tau\to\tau}$. The PCF interpreter will halt on the first term and diverge on the second. In fact, most interpreters for functional languages are "lazy," stopping at λ-abstractions and printing some message indicating that the computation will proceed no further (*e.g.*, LISP [15].)

If we observe termination at higher type, $\mathcal{A}_{\vee}$ fails to be adequate since the meanings of the two terms above are both $\perp$. To regain adequacy, one could change the interpreter to reduce inside λ-abstractions; Wadsworth [16] and Cosmadakis and

*Both authors were supported in part by NSF Grant No. 8511190-DCR, ONR grant No. N00014-83-K-0125, and NSF Graduate Fellowships.

Meyer [4, 8] give examples of such interpreters. We take the opposite approach and try to build a model that reflects the behavior of the interpreter. We are willing to add new constants to PCF, as long as we do so conservatively; the interpreter's behavior should not change on terms without the new constants, and should still stop on abstractions.

We choose "termination of closed terms at any type" and "evaluation to ground constants" as the fundamental observations. We introduce the model $\mathcal{A}_{\mathsf{Y}}$, built using the common domain-theoretic constructor of lifting, which includes an extra element at every functional type. The extra element is precisely what we need to give distinct values to $\lambda x^{\tau}.\Omega^{\tau}$ and $\Omega^{\tau\to\tau}$. We show that $\mathcal{A}_{\mathsf{Y}}$ is adequate, and with the addition of pcond (at all types) and a convergence testing operator up?, the model becomes fully abstract for our notion of observation.[1]

In $\mathcal{A}_{\mathsf{Y}}$, there is a natural way to select a set of **computable** values from the domains.[2] PCF terms always have computable meanings, but the computable values may not all be programmable. We say that a language is **universal** for a denotational semantics iff all computable semantic values are definable [11].

In PCF, all computable first-order functions are definable; these are precisely the partial recursive functions on integers. However, there are many higher-order computable functions which cannot be defined even when the language is extended with up? and pcond. One of them is a continuous approximation to the existential quantifier $\exists$ [11]. As in [11], this is essentially the only function missing; once $\exists$ has been added, the language becomes universal for the model $\mathcal{A}_{\mathsf{Y}}$.

Adequacy, full abstraction and universality mark an intimate connection between PCF and the model $\mathcal{A}_{\mathsf{Y}}$. The point of obtaining such a model is, in part, to develop techniques for proving properties about code. We give some preliminary results in defining a logic (based on LCF [6, 13]) for a fragment of PCF with up?. The logic is shown to be sound for the model $\mathcal{A}_{\mathsf{Y}}$.

[1] These results were obtained independently from Abramsky [1] and Ong [9, 10], who have proven similar adequacy and full abstraction results for an *untyped* λ-calculus. Cosmadakis [4] has extended our results to a language with product, sum, and recursive types.

[2] Every isolated element [14] in the model may be given a Gödel number n; an arbitrary element d is computable if $\{n : e_n \text{ is isolated and } e_n \sqsubseteq d\}$ is r.e.

$$
\begin{array}{rcl}
(\lambda x.M)N & \to & M[x := N] \\
\mathsf{succ}\, n & \to & n+1 \\
\mathsf{pred}\, n & \to & n\dot{-}1 \\
\mathsf{zero?}\, 0 & \to & \mathtt{tt} \\
\mathsf{zero?}\, (n+1) & \to & \mathtt{ff} \\
\mathsf{cond}\, \mathtt{tt}\, M\, N & \to & M \\
\mathsf{cond}\, \mathtt{ff}\, M\, N & \to & N \\
\mathbf{Y}M & \to & M(\mathbf{Y}M)
\end{array}
$$

$$\frac{M \to M'}{MN \to M'N}$$

$$\frac{N \to N',\ c \in \{\mathsf{pred}, \mathsf{succ}, \mathsf{cond}, \mathsf{zero?}\}}{cN \to cN'}$$

Figure 1: *Operational Rules for PCF*

2 Review of PCF

The language PCF is simply-typed λ-calculus, with types given by the grammar

$$\sigma ::== \iota \mid o \mid \sigma \to \sigma$$

The type constants ι and o are used for integers and Booleans respectively. Structured rewrite rules for the interpreter are given in Figure 1. We write $M \twoheadrightarrow N$ when M reduces to N in zero or more steps of evaluation. A term M is **stopped** if it cannot be rewritten further. For example, $\lambda x.\mathsf{succ}\, 3$ does not rewrite further, despite the fact that it has a redex as a subterm. This is essentially the language given in [11]. The main difference, aside from notation, is that $\mathsf{pred}\, 0 \to 0$ rather than stopping.

3 The Model

Our model of PCF, $\mathcal{A}_{\mathsf{Y}}$, is based on Scott domains [14] as is the standard model $\mathcal{A}_{\mathsf{V}}$. The base types are the same in both models, with $\mathcal{A}_{\mathsf{V}}[\![\iota]\!] = D^{\iota} = \{\bot, 0, 1, 2, \ldots\}$ and $\mathcal{A}_{\mathsf{V}}[\![o]\!] = D^{o} = \{\bot, \mathtt{tt}, \mathtt{ff}\}$ ordered $\bot \sqsubseteq x$ for all x. The difference between the two models appears at higher type; in $\mathcal{A}_{\mathsf{V}}$, the functional types are

$$\mathcal{A}_{\mathsf{V}}[\![\sigma \to \tau]\!] = \mathcal{A}_{\mathsf{V}}[\![\sigma]\!] \xrightarrow{c} \mathcal{A}_{\mathsf{V}}[\![\tau]\!]$$

where $D \xrightarrow{c} E$ is the cpo of continuous functions from D to E ordered pointwise [11, 13]. In $\mathcal{A}_{\mathsf{Y}}$, we lift each function space once:

$$\mathcal{A}_{\mathsf{Y}}[\![\sigma \to \tau]\!] = D^{\sigma\to\tau} = \left(\mathcal{A}_{\mathsf{Y}}[\![\sigma]\!] \xrightarrow{c} \mathcal{A}_{\mathsf{Y}}[\![\tau]\!]\right)_{\bot}$$

If D is a domain, $(D)_\perp$ is D with a new bottom element added [1, 9, 10]. Concretely, the elements of $(D)_\perp$ are $\{\langle d,0\rangle : d \in D\} \cup \{\perp\}$, ordered with $\perp \sqsubseteq d$ for all d, and $\langle d,0\rangle \sqsubseteq \langle d',0\rangle$ iff $d \sqsubseteq d'$. The function $\Uparrow : D \to (D)_\perp$ with $\Uparrow d = \langle d,0\rangle$ is an injection; the function $\Downarrow : (D)_\perp \to D$, with $\Downarrow \langle d,0\rangle = d$ and $\Downarrow\perp = \perp$, is the corresponding projection.

Given these elements, we assign meanings to terms using an *environment model* [2, 5, 7] in the usual way. Constants of base type mean the obvious elements in the domains, and constants of higher type mean lifted functions. The equations

$$\begin{aligned} \mathcal{A}_\Upsilon[\![MN]\!]\rho &= \Downarrow(\mathcal{A}_\Upsilon[\![M]\!]\rho)\,(\mathcal{A}_\Upsilon[\![N]\!]\rho) \\ \mathcal{A}_\Upsilon[\![\lambda x.M]\!]\rho &= \Uparrow f, \end{aligned}$$

where $f(d) = \mathcal{A}_\Upsilon[\![M]\!](\rho[x \mapsto d])$, specify the meanings of applications and abstractions.

4 Adequacy, Full Abstraction, and Universality

Having defined the model $\mathcal{A}_\Upsilon$ that distinguishes Ω and $\lambda x.\Omega$, we may ask to what extent the operational semantics and the model agree. A first criterion is **adequacy** [3, 8, 11]: the semantics should predict the observational outcome of interpreting a term. We have chosen to observe closed terms evaluating to a numeral at base type, and halting at higher type. Denotationally, this corresponds to meaning a number at base type, and meaning anything but $\perp$ at higher type. For our notion of observation, $\mathcal{A}_\Upsilon$ is an adequate model:

Theorem 1 (Adequacy) *The lifted model $\mathcal{A}_\Upsilon$ is adequate for PCF with respect to observing numerals and termination, i.e., for closed terms M, integers n, and proper Booleans b,*

$$\begin{array}{lll} \mathcal{A}_\Upsilon[\![M]\!]\rho = n & \text{iff} & M \twoheadrightarrow n \\ \mathcal{A}_\Upsilon[\![M]\!]\rho = b & \text{iff} & M \twoheadrightarrow b \\ \mathcal{A}_\Upsilon[\![M]\!]\rho \neq \perp & \text{iff} & \text{evaluation of } M \text{ halts.} \end{array}$$

A fully abstract model allows one to substitute denotational reasoning for operational reasoning.

Definition 1 *A denotational semantics $[\![\cdot]\!]$ is* fully abstract *(with respect to a set of observations) if for any terms M, N, $[\![M]\!] = [\![N]\!]$ iff M and N are observationally congruent.*

Plotkin [11] and Sazonov [12] show that $\mathcal{A}_\vee$ is *not* fully abstract: PCF lacks parallel facilities present in the cpo semantics, facilities that can make distinctions between observationally congruent terms. The same is true of $\mathcal{A}_\Upsilon$; it also contains parallel elements.

One way to achieve full abstraction is to extend the language. We add a parallel conditional operator $\mathsf{pcond}_\sigma : o \to \sigma \to \sigma \to \sigma$ for all types σ, with the reduction rules (*cf.* [11])

$$\begin{aligned} \mathsf{pcond}_\sigma\ \mathsf{tt}\ M\ N &\to M \\ \mathsf{pcond}_\sigma\ \mathsf{ff}\ M\ N &\to N \\ \mathsf{pcond}_\sigma\ B\ c\ c &\to c, \text{ where } \sigma = o, \iota \\ (\mathsf{pcond}_{\alpha\to\beta}\ B\ M\ N)\ Q &\to \\ & \mathsf{pcond}_\beta\ B\ (M\ Q)\ (N\ Q) \end{aligned}$$

$$\frac{B \to B'}{\mathsf{pcond}_\sigma\ B\ M\ N \to \mathsf{pcond}_\sigma\ B'\ M\ N}$$

$$\frac{M \to M'}{\mathsf{pcond}_\sigma\ B\ M\ N \to \mathsf{pcond}_\sigma\ B\ M'\ N}$$

$$\frac{N \to N'}{\mathsf{pcond}_\sigma\ B\ M\ N \to \mathsf{pcond}_\sigma\ B\ M\ N'}$$

But even with this addition, $\mathcal{A}_\Upsilon$ still makes too many distinctions between terms:

Theorem 2 *The model $\mathcal{A}_\Upsilon$ is not fully abstract for PCF+*pcond *when observing termination.*

The reason for this failure is that PCF cannot itself make all of our observations. It can observe numerals, in the sense that there is a term T_n such that $T_n M \twoheadrightarrow \mathsf{tt}$ iff M satisfied the observation "evaluates to n." However, one can show that there is no such PCF-definable test for convergence at higher type.

The solution is simple; we add convergence testing (*cf.* [1, 9, 10]) to the language. At every type, we add the operator up? with the rules

$$\begin{aligned} \mathsf{up?}\ c &\to \mathsf{tt} \\ \mathsf{up?}\ (\lambda x.M) &\to \mathsf{tt} \end{aligned}$$

$$\frac{M \to M'}{\mathsf{up?}\ M \to \mathsf{up?}\ M'}$$

Theorem 3 (Full Abstraction) *$\mathcal{A}_\Upsilon$ is fully abstract for PCF+*pcond + up? *when observing termination.*

In order to achieve universality for $\mathcal{A}_\Upsilon$, an existential quantifier, which introduces unbounded parallelism into the interpreter, must be added to PCF [11].

Theorem 4 (Universality) *PCF with the operators* pcond, up?, *and $\exists$ is universal for $\mathcal{A}_\Upsilon$.*

5 Logic for Lifted PCF

The adequacy and full abstraction theorems show that $\mathcal{A}_\Upsilon$ is a suitable guide for developing reasoning principles for code. A logic based on $\mathcal{A}_\Upsilon$ should prove *inequations* between terms rather than equations. The constant cond also requires *reasoning by cases, viz.*, if an inequation is true when a Boolean term is tt, ff, or Ω, the inequation should hold.

The wffs in the logic have the form $P \vdash M \sqsubseteq N$, where P is a set of inequations (*cf.* [13]). We write $P \vdash M = N$ as shorthand for $P \vdash M \sqsubseteq N$ and $P \vdash N \sqsubseteq M$. Due to a lack of space, we give two examples of axioms rather than the full logic:

$$\varnothing \vdash M \sqsubseteq \lambda x.M\ x$$
$$\varnothing \vdash M \sqsubseteq \mathsf{cond}\ (\mathsf{up?}\ M)\ M\ N$$

(The first resembles η-reduction [2]; note that the rule $\varnothing \vdash \lambda x.M\ x \sqsubseteq M$ is not sound, however, since it is *not* the case that $\mathcal{A}_\Upsilon[\![\lambda x.\Omega\ x]\!] \sqsubseteq \mathcal{A}_\Upsilon[\![\Omega]\!]$.) One can then show the following about the logic:

Theorem 5 (Soundness) *If* $\varnothing \vdash M \sqsubseteq N$, *then* $\mathcal{A}_\Upsilon[M] \sqsubseteq \mathcal{A}_\Upsilon[N]$.

The converse necessarily fails—the set of true inequations is not axiomatizable [13].

6 Acknowledgments

We thank Albert Meyer and Stavros Cosmadakis for helpful discussions, and Alan Fekete for reading an early draft of this paper.

References

[1] S. Abramsky. The lazy lambda calculus. December 17, 1987. Imperial College of Science and Technology. Unpublished manuscript.

[2] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Volume 103 of *Studies in Logic*, North-Holland, 1981. Revised Edition, 1984.

[3] B. Bloom. Can LCF be topped? In 3rd *Symp. Logic in Computer Science*, pages 282–295, IEEE, 1988.

[4] S. Cosmadakis. Computing with recursive types. In 4^{th} *Symposium on Logic in Computer Science*, IEEE, 1989. To appear.

[5] H. Friedman. Equality between functionals. In R. Parikh, editor, *Logic Colloquium, '73*, pages 22–37, Volume 453 of *Lect. Notes in Math.*, Springer-Verlag, 1975.

[6] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanical Logic of Computation.* Volume 78 of *Lect. Notes in Computer Sci.*, Springer-Verlag, 1979.

[7] A. R. Meyer. What is a model of the lambda calculus? *Information and Control*, 52:87–122, 1982.

[8] A. R. Meyer. Semantical paradigms: notes for an invited lecture, with two appendices by Stavros Cosmadakis. In 3rd *Symp. Logic in Computer Science*, pages 236–255, IEEE, 1988.

[9] C. L. Ong. Fully abstract models of the lazy lambda calculus. In 29th *Symp. Foundations of Computer Science*, pages 368–376, IEEE, 1988.

[10] C. L. Ong. *The Lazy Lambda Calculus: An Investigation into the Foundations of Functional Programming.* Ph.D. thesis, Imperial College, University of London, 1988.

[11] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Sci.*, 5:223–257, 1977.

[12] V. Sazonov. Expressibility of functions in D. Scott's LCF language. *Algebra i Logika*, 15:308–330, 1976. (Russian).

[13] D. Scott. A type theoretical alternative to CUCH, ISWIM, OWHY. 1969. Manuscript, Oxford Univ.

[14] D. Scott. Data types as lattices. *SIAM J. Computing*, 5:522–587, 1976.

[15] G. L. Steele. *Common Lisp: The Language.* Digital Press, Bedford, MA, 1984.

[16] C. Wadsworth. *Semantics and pragmatics of the lambda calculus.* Ph.D. thesis, University of Oxford, 1971.

[17] C. Wadsworth. The relation between computational and denotational properties for Scott's D_∞ models. *SIAM J. Computing*, 5(3):488–521, 1976.

–Cambridge, Massachusetts
April 10, 1989

DELTA: a Deduction system integrating Equational Logic and Type Assignment

V. Manca(*), A. Salibra(*) and G. Scollo(**)

(*) University of Pisa - Dip. Informatica
Corso Italia 40, I-56100 Pisa, Italy
e-mail: mancav @ dipisa.uucp, and: salibra @ dipisa.uucp

(**) University of Twente - Dept. Informatica
P.O. Box 217, NL-7500AE Enschede, Netherlands
e-mail: pippo @ utinul.uucp, or: scollo @ henut5.earn

1. Introduction

Many-sorted (conditional) equational logic is the most established basis to the algebraic approach to abstract data type (ADT) specification (see *e.g.* [EM 85]). However this logic proves somewhat inadequate in many practical situations, e.g. it entails writing a large amount of equations to deal with error cases and partially defined functions. Order-sorted algebras [G 78] were proposed in order to overcome such practical inadequacies. Nonetheless the order-sorted approach is not sufficiently flexible to deal with some aspects of ADT specification (see [M 88] for a technically detailed criticism and [P 88] for a more flexible approach to sort ordering and dependent types).

The Δ logic presented in this paper is a generalization of many-sorted equational logic that extends 'reasoning with equations' towards 'reasoning with equations and type assignments'. It provides a single, unified framework capable to cope with diverse phenomena such as *partiality*, *polymorphism* and *dependent types*. In Section 2 we illustrate and support this claim by simple examples. Section 3 is an overview of formal definitions and results. Here we summarize the main intuitions behind this logic.

1. Elements and sorts (or types, which from now on we use synonymously) are *merged* in a single carrier equipped with a binary *typing relation*, which assigns types to elements (hence types are elements themselves). This immediately introduces partiality because, in general, an operation is defined only on elements of suitable types. Moreover, one gets a great amount of flexibility and generality: several types may be assigned to an element, operations may take type arguments or yield types, etc.

2. Usual ADT presentations consist of two parts: a static one which defines the signature and a dynamic one which presents the axioms. A Δ presentation, in a more general way, *merges* the type constraints and the equality ones. In fact, Δ formulæ are conditional formulæ where equations and type assignments may occur indifferently in the premise and in the conclusion.

These intuitions were first exploited in [MS 88]: the typed equational logic introduced there is an extension of many-sorted equational logic exactly in the sense mentioned above, and soundness, completeness, and initiality results were established for it. The semantics was set in a partial-algebraic framework [ABN 80]. The pragmatics of that logic were further investigated in [MSS 88], where we also addressed the pragmatic question of how to cater, in that framework, for functions that are partially defined but non-strict (*if_then_else_* is a typical example of such a function). We found that the typing relation may offer a correctness tool, in the sense, for instance, that one may view as *meaningless terms* - but now we could otherwise say: terms representing underdefined elements, see below - those terms to which no type can ever be assigned (in a given presentation). We also noted that type as a 'correctness tool' is a concept that appears at the early days of mathematical logic (*e.g.* Russell).

Solicited by an anonymous referee, and inspired by Mosses' *Unified Algebras* [M 88], we reconsider our former enthusiasm for partial algebras under a more critical light, coming to the conclusion that, to offer an adequate representation of partiality, one need not necessarily embark on the semantical complications of the theory of partial algebras (we refer the reader to the 'Introduction' in [R 87] for a concise summary of those complications). The step of the present work from our former approach is precisely this: replacing, in the general case rather than in special ones, the syntax-sided notion of meaningless term with the semantics-sided notion of *underdefined element* ("ideal element", following Hilbert). This amounts to choose as

semantical framework *total*, rather than partial, one-sorted algebras, yet still equipped with a binary typing relation. More precisely, we will consider any element of the single carrier of any such algebra to be underdefined if *neither* any type is assigned to it *nor* is itself a type assigned to some element of the carrier. In addition to the advantages that follow from the greater simplicity of a total algebra framework, a further gain seems to be available on the methodological side too, in connection with formal notions of *refinement* and *implementation* of specifications: elements that are underdefined at a certain stage of a software engineering process may *become defined* at a later, less abstract stage - *e.g.* when some specific classification of exceptions is desired. The first example below illustrates the *exception by default* principle, useful at the more abstract stages of software design.

Two more examples illustrate, in complementary cases, the natural place that *generality* of description finds in our framework. As a matter of fact, we find that both *type polymorphism* (parameterization by types) and *dependent types* (parameterization by values) are representatives of the same species: functional abstraction. The freedom of *term construction* as a facility to express types demolishes the syntactical barriers that in ad-hoc approaches make a uniform treatment so difficult to achieve. Due to space limitations, the examples of the next Section have austere explanations and the formal overview of Section 3 gives just essential definitions and results with no proof. The full paper [MSS 89] is committed to a duly comprehensive treatment, where also the further results and investigations mentioned in Section 4 are argued in technical detail.

2. DELTA specification examples

Why yet another logic? The answer takes here the form of a few simple examples. Our bare syntax is as follows: essentially, a Δ specification is a named Δ presentation (see definitions 3.6 and 3.7) using declared variables, with the smallest (one-sorted, ranked: see definition 3.1) signature that is compatible with the axioms of the presentations.

The theory of ADT's is often identified with the theory of stacks, due to the popularity of the stack data type as specification example. The basic trouble is found here in determining which outcome should be expected from popping or topping the empty stack. With the following specification (on the left hand side) the terms pop(empty) and top(empty), among others, denote underdefined elements because they occur in no type assignment of the STACK Δ theory.

```
spec    STACK
var     s, i
in      empty : stack
        s : stack, i : item  →  push(s,i) : stack
        s : stack, i : item  →  pop(push(s,i)) ≡ s
        s : stack, i : item  →  top(push(s,i)) ≡ i
end
```

```
spec    IDENTITY (type)
var     x, d, c, f, t
in      d : type, c : type  →  d to c : type
        x : d, f : d to c  →  apply(f,x) : c
        t : type  →  id t : t to t
        t : type, x : t  →  apply(id t, x) ≡ x
end
```

The identity function is a well-known example of higher-order polymorphic function: in the example on the right hand side above, a generic 'type' parameter is declared, to enable one to specialize the definition as desired. For instance, when using such a definition in the context of a functional programming language, the parameter is to be instantiated by the (higher-order) type of the basic types of that language. Note that the syntax is assumed to allow both binary infix and unary prefix operators. Somewhat relating to the previous example, we show a use of dependent types in our last example, which can be compared with the similar example in [P 88], slightly but necessarily less parsimonious - in our opinion (argumentation in [MSS 89]).

```
spec    CATEGORY (obj, hom)
var     x, y, z, w, f, g, h
in      f : hom(x,y)  →  dom f : obj                          f : hom(x,y)  →  cod f : obj
        f : hom(x,y)  →  dom f ≡ x                            f : hom(x,y)  →  cod f ≡ y
        x : obj  →  id x : hom(x,x)                           f : hom(x,y), g : hom(y,z)  →  f ; g : hom(x,z)
        f : hom(x,y)  →  (id x) ; f ≡ f                       f : hom(x,y)  →  f ; (id y) ≡ f
        f : hom(w,x), g : hom(x,y), h : hom(y,z)  →  (f ; g) ; h ≡ f ; (g ; h)
end
```

3. Overview of DELTA

3.1 Definition Let Ω be a one-sorted algebraic signature, *i.e.* a set of operators each with a number specifying its arity. A Δ Ω-algebra $\boldsymbol{A}$ is a pair $<A, :_A>$, with A a one-sorted (total) Ω-algebra and $:_A$ (the *typing*) a binary relation on the carrier A of A.◊

3.2 Definition A Δ *morphism* from a Δ Ω-algebra $\boldsymbol{A}$ into a Δ Ω-algebra $\boldsymbol{B}$ is a morphism $\phi: A \to B$ that respects the typing, *i.e.* such that if $a_1 :_A a_2$ then $\phi(a_1) :_B \phi(a_2)$. ◊

3.3 Definition A Δ *congruence* on a Δ algebra $\boldsymbol{A}$ is a pair $\theta = <\equiv_\theta, :_\theta>$ of binary relations on A such that:

(i) $\equiv_\theta$ is a congruence on A;

(ii) if $a_1 \equiv_\theta b_1$ and $a_1 :_\theta c$, then $b_1 :_\theta c$;

(iii) if $a_1 \equiv_\theta b_1$ and $c :_\theta a_1$, then $c :_\theta b_1$

(iv) $:_A \subseteq :_\theta$. ◊

3.4 Definition If $\theta = <\equiv_\theta, :_\theta>$ is a Δ congruence on $\boldsymbol{A}$, then we let $[a]_\theta$ denote the congruence class $[a]_{\equiv_\theta}$ and define the Δ *quotient* $\boldsymbol{A}/\theta$ to be the Δ algebra $<A/\equiv_\theta, :_{A/\theta}>$ where the typing relation is defined by $[a]_\theta :_{A/\theta} [b]_\theta$ iff there exist $a' \in [a]_\theta$ and $b' \in [b]_\theta$ such that $a' :_\theta b'$. ◊

3.5 Definition $\boldsymbol{T}_\Omega(V) =_{def} <T_\Omega(V), \emptyset>$ is the *term* Δ *algebra* of signature Ω and variables V, where $T_\Omega(V)$ is the standard term algebra. ◊

3.6 Definition

atomic Δ *formulæ*:	(i)	$t1 \equiv t2$	(equations)		
	(ii)	$t1 : t2$	(type assignments)	with $t1, t2 \in T_\Omega(V)$,	
Δ *formulæ*:	(iii)	$\Gamma \to \alpha$			

with α an atomic Δ formula, called *conclusion*, and Γ a finite, possibly empty set of atomic Δ formulæ, called *assumption*. ◊

3.7 Definition A Δ *presentation* is a triple $<\Omega, V, E>$, where E is a finite set of Δ formulæ on Ω and V. ◊

Substitution, assignment, evaluation have the usual definitions. By the evaluation lemma (existence of a unique Δ morphism extending a given assignment) term evaluation is determined by assignment. Δ *satisfaction* is then defined as one expects. The Δ *calculus* $\vdash_\Delta$ is a binary relation between Δ presentations and Δ formulæ that is constructed using two axiom schemas and eight inference rule schemas (collectively termed *rules* of the Δ calculus, for short) in the usual, proof-theoretic way. The rules are presented in Table 1, where, understanding the signature Ω and variables V (as we will often feel free to do), we adopt the following notation: (i) t, u (possibly with subscripts) are terms, (ii) α, β are atomic formulæ, (iii) Γ is an assumption, (iv) ϕ is a formula, (v) σ is a substitution : $T_\Omega(V) \to T_\Omega(V)$, extended to formulæ in the usual way, (vi) ω is a k-ary operator. ◊

1.	$E \vdash_\Delta \{\alpha\} \to \alpha$	*Tautology*
2.	*If* $E \vdash_\Delta \Gamma \to \alpha$ *then* $E \vdash_\Delta \Gamma \cup \{\beta\} \to \alpha$	*Monotonicity*
3.	$E \vdash_\Delta t \equiv t$	*Reflexivity*
4.	*If* $E \vdash_\Delta \Gamma \to t_1 \equiv t_2$ *then* $E \vdash_\Delta \Gamma \to t_2 \equiv t_1$	*Symmetry*
5.	*If* $E \vdash_\Delta \Gamma \to t_1 \equiv t_2$ *and* $E \vdash_\Delta \Gamma \to t_2 \equiv t_3$ *then* $E \vdash_\Delta \Gamma \to t_1 \equiv t_3$	*Transitivity*
6.	*If* $E \vdash_\Delta \Gamma \to \alpha$ *then* $E \vdash_\Delta \sigma(\Gamma) \to \sigma(\alpha)$	*Substitution*
7.	*If* $E \vdash_\Delta \Gamma \to t_i \equiv u_i$ $(i=1,...,k)$ *then* $E \vdash_\Delta \Gamma \to \omega(t_1,...,t_k) \equiv \omega(u_1,...,u_k)$	*Replacement*
8.	*If* $E \vdash_\Delta \Gamma \cup \{\alpha\} \to \beta$ *and* $E \vdash_\Delta \Gamma \to \alpha$ *then* $E \vdash_\Delta \Gamma \to \beta$	*Modus Ponens*
9.	*If* $E \vdash_\Delta \Gamma \to t_1 \equiv t_2$ *and* $E \vdash_\Delta \Gamma \to t_1 : u$ *then* $E \vdash_\Delta \Gamma \to t_2 : u$	*Typing equals*
10.	*If* $E \vdash_\Delta \Gamma \to u_1 \equiv u_2$ *and* $E \vdash_\Delta \Gamma \to t : u_1$ *then* $E \vdash_\Delta \Gamma \to t : u_2$	*Equating types*

Table 1 : The rules of the Δ calculus

3.8 Proposition The Δ calculus is *sound*: if the Δ algebra $\mathbf{A}$ satisfies the presentation E, then it satisfies any formula derivable from E; in symbols: $(\mathbf{A} \models E \wedge E \vdash_\Delta \phi) \Rightarrow \mathbf{A} \models \phi$. ◊

3.9 Theorem The Δ calculus is *complete*: if the formula ϕ is a logical consequence of the presentation E, then it is derivable from E; together with the soundness (proposition 3.8), this is formulated as: $E \vdash_\Delta \phi \Leftrightarrow E \models \phi$. ◊

Let T_Ω denote the ground term Δ Ω-algebra; T_Ω/E is then the quotient of T_Ω by the Δ congruence defined via the Δ calculus.

3.10 Theorem T_Ω/E is *initial* in the class of Δ Ω-algebras that satisfy E. ◊

4. Summary of further results, current work and future developments

Further results have been obtained in [MSS 89] relating to representation in Δ of order-sorted logic [G 78] and of the logic of partial algebras [B 86], [BW 82]. For instance, if O is an order-sorted presentation and ϕ a formula of that logic, then $O \models_{os} \phi$ iff $\tau_\Delta(O) \vdash_\Delta \tau_\Delta(\phi)$, where τ_Δ is a suitable translation operator. In a similar manner Δ enables one to obtain calculi and completeness theorems for the logic of partial algebras, as well as other logics (category theory). On the computational side, generalizations of the confluence results presented in [BK 86] are available for Δ.

We are currently studying the potential of Δ for applications, *e.g.* specification of software systems: notions of hierarchy and modularity are here the main topics of investigation. Some future work will be concerned with a particular, especially intriguing application domain: the algebraic formulation of significant fragments of natural language grammars. We dared a glimpse at this area in our previous work [MSS 88] and were encouraged for a great ease of expression, which ensues with integrating equality, types and term construction.

Essential References

[ABN 80] H. Andreka, P. Burmeister and I. Nemeti, Quasivarieties of partial algebras - a unifying approach towards a two-valued model theory for partial algebras, Preprint Nr. 557, FB Mathematik und Informatik, TH Darmstadt, 1980.

[B 86] P. Burmeister, A model theoretic oriented approach to partial algebras, Akademie-Verlag, Berlin, 1986.

[BK 86] J.A. Bergstra and J.W. Klop, Conditional rewrite rules: confluence and termination, JCSS 32, 3 (1986) 323-362.

[BW 82] M. Broy, M. Wirsing, Partial Abstract Types, Acta Informatica 18 (1982) 47-64.

[EM 85] H. Ehrig, B. Mahr, Fundamentals of Algebraic Specification 1, Springer-Verlag, Berlin, 1985.

[G 78] J.A. Goguen, Order Sorted Algebra, Semantics and Theory of Computation R. 14, UCLA C.S.Dept., 1978.

[M 88] P.D. Mosses, Unified Algebras and Modules, DAIMI PB-266, Univ. Aarhus, C.S.Dept., Oct.1988; ACM POPL '89.

[MS 88] V. Manca and A. Salibra, On the power of equational logic: applications and extensions, Proc. Int.l Conf. on Algebraic Logic, Budapest, August 8-14, 1988 (to appear).

[MSS 88] V. Manca, A. Salibra and G. Scollo, On the nature of TELLUS, Memo. INF-88-57, Univ. Twente, NL, Dec.1988.

[MSS 89] V. Manca, A. Salibra and G. Scollo, Equational Type Logic, Draft, Univ. Pisa, I, & Univ. Twente, NL, Apr.1989.

[P 88] A. Poigné, Partial Algebras, Subsorting and Dependent Types, in: D. Sannella, A. Tarlecki (Eds.), Recent Trends in Data Type Specification, Springer-Verlag LNCS 332 (1988) 208-234.

[R 87] H. Reichel, Initial Computability, Algebraic Specifications, and Partial Algebras, Oxford University Press, 1987.

On Algebraic Transformations of Sequential Specifications

R. Janicki

Department of Computer Science
and Systems
McMaster University
1280 Main Street West
Hamilton, Ont., Canada L8S 4K1

T.Müldner

Visiting Professor
Department of Computer Science
The University of Western Ontario
London, Ont., Canada N6A 5B7

* On sabbatical leave from:
Jodrey School of Computer Science
Acadia University, Wolfville, N.S.

In this paper we show an algebraic transformation of sequential specifications to the equivalent concurrent specifications. Here, we consider sequential specifications in the form of regular expressions extended with a declaration of the actions that are independent and have a potential for a concurrent execution. This kind of a sequential specification can be represented in the **sequential** programming language, called Banach. (Banach has been designed by us in such a way that the programmer does not have to be concerned about synchronization details, [JM88, JM89].) The concurrent specification can be translated into an equivalent concurrent specification, and finally into a concurrent programming language, such as occam.

The above results have important applications in *software technology*. The user of the Banach programming language can take advantage of the increased efficiency of concurrent architectures, and at the same time she/he can concentrate on algorithms being implemented and disregard technical issues, such as low-level synchronization details. The (automatic) transformation of the provided sequential specification will yield an equivalent concurrent specification. This approach has its origin in research described in [J81, LH82.]

A sequential specification of executions of actions from some alphabet A is given by a regular expression R. The semantics of this specification is defined by two components ([J85, JL88]): the set RFS(R) of **resulting histories** of R defined as the language generated by the expression R, and the set FS(R) of **histories**, (or, firing sequences), defined as Pref(RFS(R)); where for a language $L \subset A^*$, $Pref(L) = \{x \in A^*: \exists y \in A^* (xy \in L)\}$.

Concurrent regular expression is of the form $CR = R_1 || \ldots || R_n$, where for $i=1,\ldots,n$; R_i is a regular expression. The semantics of concurrent expressions can be defined in an algebraic way, using vector sequences [Shi79]. First, for any action $x \in A$, where A is the alphabet of CR defined as the union of alphabets of component expressions R_i, we denote by $\underline{x}$ the vector $[h_1(x),\ldots,h_n(x)]$, where $h_i(x)$ is a if $x \in A_i$, and ε otherwise (here, ε is a distinguished element denoting null.) Then we put $vect(L) = \{\underline{x}: x \in L\}$, for any $L \subset A^*$. Now, the semantics of CR is defined by the set of **resulting histories**

$RVFS(CR) = vect(A^*) \cap RFS(R_1) \times \ldots \times RFS(R_n)$

and the set of **histories**

$VFS(CR) = vect(A^*) \cap FS(R_1) \times \ldots \times FS(R_n)$.

Both these sets are closed under the operation Pref.

Note that the above semantics, described in terms of vector firing sequences, can be equivalently described using Mazurkiewicz's traces (see [Maz77, Maz86]):

For $u,v \in A^*$, a **shuffle** of u and v is defined as

$$sh(u,v)=\{u_1v_1u_2...u_nv_n:\ u=u_1u_2...u_n,\ v=v_1v_2...v_n \text{ for } i = 1,...n,\ u_i \in A^*,\ v_i \in A^*\}.$$

For $L_1, L_2 \subset A^*$, we put

$$sh(L_1, L_2) = \bigcup_{\{u \in L_1, v \in L_2\}} sh(u,v).$$

Now, we define a **parallel composition** of words and languages. For a partition A_1, A_2 of A and $u \in A_1^*$, $v \in A_2^*$ we put

$$u||v = sh((A_2-A_1)^*,u) \cap sh((A_1-A_2)^*,v)$$

and for $L_1 \subset A_1^*$, $L_2 \subset A_2^*$ we put

$$L_1||L_2 = \{x: x = u||v,\ u \in L_1,\ v \in L_2\}.$$

By associativity, we extend the operation || to n words and n languages, $n \geq 2$. Then, for a CR:$R_1||...||R_n$ we can define the set of resulting histories as a parallel composition of resulting histories of components:

$$RFS(R_1)||...||RFS(R_n)$$

and similarly, we define the set of histories as a parallel composition of histories of components:

$$FS(R_1)||...||FS(R_n).$$

By a **p-concurrent regular expression** (a *potentially concurrent* expression) we mean a regular expression R, the alphabet of which is partitioned into a finite number of subsets (intuitively, actions that are mutually dependent occur in the same subset.) Thus, a p-concurrent regular expression, (PCR) is a pair $(R, A=A_1 \cup A_2 \cup ... \cup A_n)$ where A is the alphabet of R. We define the semantics of p-concurrent expressions using vector firing sequences (comp. [Jan85].) The set of **resulting histories** of the p-concurrent expression is defined as

$$RVFS(CR) = vect(RFS(R))$$

and the set of **histories** of the p-concurrent expression is defined as

$$VFS(CR) = Pref(vect(RFS(R))) = Pref(RVFS(CR)).$$

Here, R is the first component of CR.

As above, the semantics of p-concurrent regular expressions can be defined using traces: Let I be an independence relation over the alphabet A of a CR: $(R, A=A_1 \cup A_2 \cup ... \cup A_n)$ defined as follows: $(u,v) \in I$ if $\forall$ i, $u \notin A_i$ or $v \notin A_i$. We denote by ind a relation over A^* associated with the relation I: $(u,v) \in ind$ if v can be obtained from u by permuting successive letters that are in the relation I. For a language $L \subset A^*$, a **trace language**, tr(L) is a set of equivalence classes L/ind of elements of L. Now, we define resulting histories as tr(RFS(R)), and histories as Pref(tr(RFS(R))).

As mentioned above, p-concurrent regular expressions have a *potential* for a concurrent execution. In order to reveal this potential, for a given p-concurrent expression PCR we should find a concurrent expression that will be equivalent to this PCR. For this sake, we now describe a **transformation** ç of p-concurrent expression PCR : $(R, A=A_1 \cup A_2 \cup ... \cup A_n)$ into concurrent expression CR of the form CR: $R_1||...||R_n$ (see [Jan85.]) Each of the component expressions is formed by erasing, or the concealment in R of these actions that do not appear in A_i. Thus, $Alpha(R_i) = A_i$ and $RFS(R_i) = h_i(RFS(R))$. The transformation ç is not a function, that is there may be more than one element of $\rho(PCR)$. From the definition of ç it follows that

$$R_1 ||...|| R_n \in ç(PCR) \text{ iff } (\forall i)\ RFS(R_i) = RFS(PCR_i^{\varepsilon})$$

where PCR_i^{ε} is derived from PCR by replacing all elements of $A-A_i$ by ε.

We say that a p-concurrent regular expression PCR = $(R, A=A_1 \cup A_2 \cup ... \cup A_n)$ is **proper** if ç(PCR) is equivalent to PCR, that is

$$VFS(PCR) = VFS(ç(PCR)) \text{ and } RVFS(PCR) = RVFS(ç(PCR)).$$

(The above equalities are well-defined because for $\xi_1, \xi_2 \in ç(PCR)$ we have $VFS(\xi_1) = VFS(\xi_2)$ and $RVFS(\xi_1) = RVFS(\xi_2)$.)

Example

PCR: a;b;c A_1={a,b} A_2={a,c} ç(PCR)∋CR: ((a;c) || (b;c)) ■

Since the computations of PCR produce the same histories and resulting histories as the computations of CR, the above PCR is proper. In general, a PCR and a resulting CR may have different sets of histories and identical sets of resulting histories, or identical sets of histories and different sets of resulting histories. An example of the former case is

PCR: (a;c;e), (b;d;f) $A_1 = \{a,e,b,f\}, A_2 = \{c,e,d,f\}$

ç(PCR)∋CR: ((a;e),(b;f)) || ((c;e),(d;f))

for which the sets of resulting histories are identical, but the set of histories of CR includes the sequence ad, (leading to a deadlock), which clearly is not a history of the PCR. An example of the latter case is

PCR: (a,b)* $A_1 = \{a\}$ $A_2 = \{b\}$ ç(PCR)∋CR: a*||b*

for which the sets of histories are identical, but resulting histories of the PCR must have the same number of occurrences of a's and b's, while resulting histories of the CR contain arbitrary number of these actions.

Note that in the above examples p-concurrent expressions were *not* proper because of the conflict between the choice constructor "," and the independency relation: independent actions occurring in branches of the choice were mapped by ç to different components of the parallel construct ||. Thus, we introduce synchronization guards which are in conflict with such actions. Synchronization guards will be inserted into alternatives and loops. Formally, let **Σ** be a class of **synchronized p-concurrent** regular expressions defined by the following grammar:

expr ::= el | el ; el
el ::= action | (Δ; expr)* | (expr) | alt (Δ is called a synchronization guard)
alt ::= expr , (Δ ; expr)

such that for i(a) = {j: a∈A_j } the following conditions are satisfied:

- for each loop (Δ;expr)* $\forall b \in Alpha(expr)\ (i(b) \subset i(\Delta))$
- for each alternative (expr1, (Δ; expr)) $\forall b \in (Alpha(expr1) \cup Alpha(expr))\ (i(b) \subset i(\Delta))$
- synchronization guards are unique

Now, we define a **transformation Π** from the set **R** of all p-concurrent regular expressions into **Σ**. The mapping Π inserts the synchronization actions as described above. Thus, the alphabet A of any expression from **Σ** is extended with a number of symbols from some alphabet SYNC, disjoint with A.

Let us explain the conditions in the definition of the class **Σ**. The first two of the above conditions state that synchronization guards are not independent with other actions in the same alternative, or loop. Note that synchronization guards are not necessarily dependent with *all* other actions. This is because we do not wish to limit concurrency by introducing synchronization guards, that is we require that the set of histories of the transformed expressions with synchronizing actions concealed should be identical to the set of histories of the original expression. For example, if

PCR: a;(b,c) A_1={a} A_2 = {b,c}

then the concurrency in the expression CR: ((a;(ε,Δ))) || ((b,(Δ;c))), resulting from the expression

Π(PCR): a;(b, (Δ ;c)) $A_1 = \{a, \Delta\}$ $A_2 = \{b, c, \Delta\}$

would be unnecessarily limited; for example the sequence ca is a history of PCR but is not a history of CR.

The reason the third condition above requires synchronization guards to be unique is explained by the following example:

PCR: a,b,c $A_1 = \{a\}, A_2 = \{b\}, A_3 = \{c\}$

Here, the expression PCR1 with a *non* -unique synchronization guard is not proper:

Π(PCR): a, (Δ;b), (Δ;c) $A_1 = \{a, \Delta\}$ $A_2 = \{b, \Delta\}$ $A_3 = \{c, \Delta\}$

but, indeed, the expression Π(PCR) with *unique* synchronization guards is proper:

Π(PCR): a, (Δ1;b), (Δ2;c) $A_1 = \{a, \Delta1, \Delta2\}$ $A_2 = \{b, \Delta1, \Delta2\}$ $A_3 = \{c, \Delta2\}$ ■

Theorem 1

Every synchronized p-concurrent regular expression from the class Σ is proper.

Let X|SYNC denotes the concealment of actions from SYNC. It can be proved that **synchronization guards do not limit potential concurrency:**

Theorem 2

For every concurrent regular expression R

VFS(R) = VFS(Π(R)) | SYNC and RFVS(R) = RVFS(Π(R)) | SYNC

Therefore, for a specification S of a sequential system in the form of a p-concurrent regular expression (which can be obtained from a Banach program), we can first apply the transformation Π to get a proper specification Π(S), and then the transformation ç, to get an equivalent concurrent specification ç(Π(S)). For example, for the expression

PCR: a,b,c, A1 = {a} A2 = {b} A3 = {c}

we have Π(R): a,(Δ1;b),(Δ2;c) A1 = {a,Δ1,Δ2} A2 = {b,Δ1,Δ2} A3 = {c,Δ2}

and ç(Π(R)) is of the form: a,Δ1,Δ2 || ε,(Δ1;b),Δ2 || ε,ε,(Δ2;c)

Acknowledgements

The work of the first author was partially supported by the NSERC grant OGP0036539 and the work of the second author was partially supported by the NSERC General Grant, Acadia University, 1987.

Bibliography

[JL88] R. Janicki, P.Lauer. *Specifications and Analysis of Concurrent Systems; The COSY Approach.* A monograph, Springer (to appear.)

[JM89] R.Janicki, T.Müldner. Complete Sequential Specifications Allows for a Concurrent Execution, *ACM 1989 Computer Science Conference*, Feb. 21-23, 1989, Louisville, Kentucky,USA

[JM88] R.Janicki, T.Müldner. Sequential Specifications and Concurrent Executions of Banach Programs. *CIPS'88*. Edmonton, Canada.

[Jan88] R.Janicki. How to relieve a Programmer from Synchronization Details. *Proc. of 16th. Annual ACM Computer Science Conference* . Feb. 23-35, 1988. Atlanta, USA.

[Jan81] R.Janicki. On the Design of Concurrent Systems. *Proc. 2nd. Conf. on Distributed Computing Systems*. Paris, 1981 (IEEE Press, New York, 1981, pp.455-466.)

[Jan85] R.Janicki. Transforming Sequential Systems into Concurrent Systems. *Theoretical Computer Science* 36 (1985), pp.25-58.

[LH82] C.Lengauer, E.C.R.Hehner. A Methodology for programming concurrency: An informal approach. *Sci. Comput. Programm.* 2 (1982), pp. 1-18.

[Maz77] A.Mazurkiewicz. Concurent Program Schema and Their Interpretations. *Report DAIMI-PB-78* , Aarhus University, 1977.

[Maz86] A.Mazurkiewicz. Trace Theory. *Lecture Notes in Computer Science* 255, Springer 1986.

[Shi79] M.W. Shields. Adequate Path Expressions. *Lecture Notes in Computer Science* 70, Springer 1979.

An Algebraically Specified Language for Data Directed Design

Eric G. Wagner
Computer Science Principles
Mathematical Sciences Department
IBM Research Division, T. J. Watson Research Center
Yorktown Heights, NY 10598 / USA

1 Introduction

This paper is a preliminary version of the background material for the talk I will be presenting at the International Conference on Algebraic Methodology and Software Technology, Iowa City, Iowa, May 22-24 1989.

For many years I have been working on algebraic/categorical methods for specifying various programming language constructs with particular emphasis on the specification of data types [10, 9], and the specification of programming languages as-a-whole [5, 8, 6, 7, 11]. In this paper I combine these interests, and present an algebraic/categorical specification of a language for specifying abstract data types. My interest goes beyond the specification of types to the more general topic of data-directed design. The key idea of data-directed design is that software design should be centered about the design of data types rather than about the design of procedures. I don't have the time, or space, here to present detailed arguments for data directed design, but Bertrand Meyer gives a good presentation of them in [2]. The basic argument is that a data directed approach supports such good things as maintainability, reusability, and understandability. The tools from data directed design that are used to realize these good things are such concepts as extensibility, encapsulation (information hiding), generic types, and inheritance.

A data type is specified in my language by giving a "program" that implements it. Thus these specification are not algebraic specifications as defined in [10, 9]. Indeed, they are, what might be called, specifications-by-example. However, they are still abstract specifications. The desired "abstraction" is achieved through encapsulating the programs so that one can only exploit WHAT the program does, and not HOW it does it. Needless to say, "encapsulating programs" is not a new idea, but what is new here is that we do it in a rigorous mathematical framework that permits analysis.

The flavor of the language is close to that of many "object-oriented languages" such as SMALLTALK or EIFFEL. In particular, we follow SMALLTALK in using the terms "class", "object", and "method". Roughly speaking, a class is a data type, an object is an

instance of a data type, and a method is an operation on a data type. However, objects, in contrast to data types, have "memory" and this means that we are outside the familiar domain of algebraic specifications. On the other hand, we are not as close to SMALLTALK as our choice of terminology might suggest.

- We do not have any built in types, not even BOOL.
- We use a different form of objects. Most object-oriented languages define objects as being Records, that, as elements of products. We define objects as being as Variants over Records, that is, as elements of a sum of products (or, more precisely, as a coproduct of products in the category of sets).
- We use a "method calling" paradigm rather than the message sending paradigm of SMALLTALK. But a method still belongs to a specific class. We permit a method belonging to a class k to access and/or modify the value of any its parameters or variables of class k.
- Associated with each class k are Case, Assignment, and object creating operations that can only be used within methods belonging to k.

Some of these differences will be motivated in more detail as we go along. For additional motivation see (or await) [11].

The outline of the paper is as follows: Section 2 gives an informal overview of the language. A very brief introduction into the algebraic specification of languages is provided in Section 3. Section 4 gives the syntax of the language. Section 5 defines the class of algebras used in the semantics which is then presented in Section 6. In Section 7 we give examples of the use of the language to define some familar data types. Section 8 takes a brief look at some of the issues I hope address more fully in my talk.

Some notation: Given a set K, we write K^* for the set of strings on K, and $(K^*)^*$ for the set of strings-of-strings on K. We write λ for the empty string in K^*, and ( ) for the empty string in $(K^*)^*$. Given strings $v_1, ..., v_n$ in K^*, we write $(v_1)\cdots(v_n)$ to denote the string in $(K^*)^*$ whose ith element is v_i. Given a string u we write $|u|$ to denote the length of u.

2 Informal Overview of the Language

This section gives an informal overview of the language. The vocabulary used is close to that used by the SMALLTALK community. But I want to warn both programmers and mathematicians that words such as *class*, and *object* may not have the meaning they might expect.

A *program* consists of a specification of a collection K of *classes*. A *class* k consists of a specification of the *form of the objects of k* together with the collection of *methods belonging to k*. An *object* in k is either, nil_k, the *nil object of k*, or it is an *instance of k*. An *instance of k* has a *value* which is a tuple of *objects*. The *form of k* specifies which tuples may occur

as values of instances of objects from k. A *method belonging to k* is a specification of an operation on *objects*. The *specification of a method σ* will specify the *parameters* of σ, the *temporary variables* used in σ, the *expression* describing the steps of σ, and the *class of the result returned by σ*. The execution of a *method of class k* will, providing it terminates, *return a result* and may change the value of some of its parameters.

> For example, we can give a program specifying the classes: BOOL, NAT, INT, STACK-OF-INT. The form an object of class STACK-OF-INT could specify that an instance of object of STACK-OF-INT will have a value that is either an empty tuple $\langle\ \rangle$, or a pair $\langle S, I\rangle$ where S is an object of class STACK-OF-INT and I is an object of class INT. The intuition is that a STACK-OF-INT is either empty or it consists of a top element I and a "substack", S, corresponding to the remainder of the stack. The class STACK-OF-INT would have methods for operations such as POP, PUSH, and MAKE-EMPTY-STACK. The method for POP would specify that it has a STACK-OF-INT as parameter, and that it returns an object of class INT. This example is worked out in detail in section 7

The form of a class k restricts the values of instances to a given sum of products of the sets of objects of specified classes. The form of a class is fixed but the specific sets will change with time – in effect, objects do not come into existence until they are needed.

> In the example of STACK-OF-INT the form will restrict the values to the set $(1 + (O_{stack} \times O_{int}))$ where 1 denotes the product of the empty set of sets (the one-element set containing the empty tuple $\langle\ \rangle$), O_{stack} is the set of objects of class STACK-OF-INT, and O_{int} denotes the set of objects of class INT.

The *expression* specifying the steps of a method σ, belonging to class k, is built from *primitive operations* together with the parameters and temporary variables of σ. The desired encapsulation of classes is achieved by restricting the writing of methods so that knowledge of the form of a class k can only be exploited within methods belonging to the class k. For each class k, we use the form of k to define a set of basic operations that can only be used within expressions specifying methods belonging to k. Briefly, for each class k we have *private operations*:

NEW(k, i), an operation that creates a new instance of an object of class k with value the all-*nil* tuple for summand i.

CASE$(e_0, e_1, \ldots, e_n)$, a case statement with a case for each of the n summands of k. If the expression e_0 evaluates to a tuple in the ith summand of k then the expression e_i is evaluated.

CHANGE$(e_0, i, e_1, \ldots, e_n)$, an operation for changing the value of an object of class k. Changes the value of the object of class k resulting from the evaluation of the expression e_0 to the n-tuple for summand i resulting from evaluating the expressions $e_1, \ldots, e_n$.

ACCESS(e_0, i, j) , an operation for accessing components of objects of class k. The operation returns the jth component of the ith summand of the object of class k resulting from evaluating the expression e_0.

In addition there are the following *public operations* that can be used in any method of any class.

NIL$_k$ a constant, denoting the *nil* object of class k – note that the *nil* objects are typed.

INST(e_0, e_1, e_2), a conditional operation, evaluates expression e_0 to get an object x of class k, then evaluates e_1 if x is an instance of k, but evaluates e_2 if x is the *nil* object of k.

ASSIGN(i, e_0), e_0 evaluates to an object of class k which is assigned to the ith temporary variable of class k.

$e_1; e_2$, an operation for composing the evaluations of expressions.

CALL$(\rho, e_1, \ldots, e_n)$, an operation for calling methods of other classes. Method ρ is called and passed, as parameters, the objects resulting from evaluating the expressions $e_1, \ldots, e_n$. A *method belonging to a class* k can only access, or change, the value of objects of a class $k' \neq k$ by *calling methods belonging to* k'.

The syntax given in section 4 ensures that the applications of these operations are well-defined.

To define the methods POP and PUSH for STACK-OF-INT we can use the NEW, CASE and CHANGE operations corresponding to the form of STACK-OF-INT. The NEW operation for STACK-OF-INT can be used to create either the empty STACK-OF-INT corresponding to the empty-tuple $\langle\ \rangle$, or to produce a pair $\langle nil_{stack}, nil_{int} \rangle$, the latter operation is not of any interest in this example. The CASE operation for STACK-OF-INT, has two cases, corresponding intuitively to empty-stack and non-empty stack. The CHANGE operation for STACK-OF-INT, allows us to change the value a STACK-OF-INT object as required by the POP and PUSH methods.

3 Algebraic Specifications of Languages

As mentioned in the introduction, I have been working on algebraic/categorical methods for specifying the design of imperative programming languages. The idea is to provide a framework for language design that is simultaneously operational, abstract, and prescriptive. By operational I mean that I can talk about executions of programs, and about operations such as declaring variables, creating pointers, assigning values, etc. By "abstract" I mean that I can describe "what" happens without saying just "how" it is done –

for example, I can talk about "declaring variables" or "creating pointers" without giving an overly specific implementation of this within some "machine". By "prescriptive" I mean that the framework naturally promotes good design and understanding of good design.

Some underlying ideas of this approach are

- to model the execution of a program in terms of state transitions where the states are algebras and represent not just "the memory" but also include "the currently declared" types, variables, pointers, constants, etc.
- that the basic operations on states should be natural categorical operations on the algebras and/or their signatures. For example, as shown in [6], declarations of variables, pointers, and data types, can all be described in terms of pushouts in an appropriate category of algebras.
- that records and variants are key concepts, that they correspond to products and coproducts and that their associated morphisms (projections, injections, and mediators) correspond to important programming concepts. For example, the mediating morphism for a coproduct representing a variant correspond to the case statements (or case expressions) used for type-safe access to the variant.

We use this approach in this paper, but, by and large, we avoid explicit mention of the categorical constructions, and present the semantic constructions without discussing the mathematical motivations behind them. However, examination will reveal that the definitions **INST**, **NEW**, **CASE**, **CHANGE** and **ACCESS** operations exploit the available categorical structure, sometimes at several levels.

Section4 gives the syntax of the language. The syntax, as given, is not very user friendly, so we follow it by some informal sugaring which we employ in the examples in Section 7. In Section 5 we describe the algebras that are used to describe the states. Finally, in Section 6 we describe the operations on the state-algebras that give the semantics of the language.

4 Abstract Syntax

A *specification of classes* consists of the following data:

K, a set (of *class names*).

Σ, a set (of *method names*).

$\alpha : \Sigma \to K \times K^* \times K$. If $\sigma \in \Sigma$, and $\alpha(\sigma) = \langle k, u, t \rangle$ then σ *belongs* to the class k, has $|u|$ *arguments* where the ith argument is of class u_i, and σ *returns* a value of class t.

$\iota : K \to (K^*)^*$. If $\iota(k) = v_1 \cdots v_n \in (K^*)^*$ with $v_i = v_{i,1} \cdots v_{i,n_i} \in K^*$, then the class k is of *form* $\iota(k)$, has n *summands* the jth of which, for $j \in \{1, \ldots, n\}$ having n_j *components* the ith of which, for $i \in \{1, \ldots, n_j\}$, being of class $v_{j,i}$.

$\tau : \Sigma \rightarrow K^*$. If $\tau(\sigma) = w$, then the method σ has $|w|$ *temporaries (local variables)*, the ith of which is of class w_i.

$\xi : \Sigma \rightarrow$*Expr*. Where $\xi(\sigma)$ is *body-expression* of the method σ. We call *Expr* the *set of expressions*. If $\alpha(\sigma) = \langle h, u, k \rangle$ then $\xi(\sigma) \in Expr_{k,\sigma}$, the set of *k-σ-expressions*, defined as follows:

$\texttt{NIL}_k$ is a k-σ-expression.

$\texttt{P}_{\sigma,i}$ is a k-σ-expression if $\alpha(\sigma) = \langle h, w, t \rangle$ and $i \in \{1, \ldots, |w|\}$ such that $w_i = k$.

$\texttt{T}_{\sigma,i}$ is a k-σ-expression if $i \in \{1, \ldots, |\tau(\sigma)|\}$ such that $\tau(\sigma)_i = k$.

$e_1 ; e_2$ is a k-σ-expression if e_1 is a j-σ-expression for some $j \in K$, and e_2 is a k-σ-expression.

$\texttt{INST}(e_0, e_1, e_2)$ is a k-σ-expression if e_0 is a j-σ-expression for some $j \in K$, and if e_1 and e_2 are k-σ-expressions.

$\texttt{ASSIGN}(i, e_1)$ is a k-σ-expression if $i \in \{1, \ldots, |\tau(\sigma)|\}$, $\tau(\sigma)_i = k$, and e_1 is a k-σ-expression.

$\texttt{CALL}(\rho, e_1, \ldots, e_p)$ is a k-σ-expression if $\rho \in \Sigma$, and there exist h and u such that $\alpha(\rho) = \langle h, u, k \rangle$, and, where $u = u_1 \cdots u_p$, we have that e_i is a u_i-σ-expression for each $i \in \{1, \ldots, p\}$.

$\texttt{NEW}(k, i)$ is a k-σ-expression if $i \in \{1, \ldots, |\iota(k)|\}$, and σ belongs to k.

$\texttt{CASE}(e_0, e_1, \ldots, e_n)$ is a k-σ-expression if, where $\alpha(\sigma) = \langle j, u, h \rangle$, there exists $j \in K$ such that e_0 is a j-σ-expression, $n = |\iota(j)|$, and, for each $i = 1, \ldots n$, e_i is a k-σ-expression. Note that, here, σ belongs to j but returns a k object.

$\texttt{CHANGE}(e_0, i, e_1, \ldots, e_p)$ is a k-σ-expression if e_0 is a k-σ-expression, there exist u and h such that $\alpha(\sigma) = \langle k, u, k \rangle$, so σ belongs to k, and, where $\iota(k) = v_1 \cdots v_n \in (K^*)^*$, we have $i \in \{1, \ldots, n\}$, $p = |v_i|$, and, $v_i = v_{i,1} \cdots v_{i,p}$ where, for each $j \in \{1, \ldots, p\}$, e_j is a $v_{i,j}$-σ-expression.

$\texttt{ACCESS}(e_0, i, j)$ is a k-σ-expression if e_0 is a k-σ-expression, there exist u and h such that $\alpha(\sigma) = \langle h, u, k \rangle$, and, where $\iota(k) = v_1 \cdots v_n \in (K^*)^*$, we have $i \in \{1, \ldots, n\}$, and, where $v_i = v_{i,1} \cdots v_{i,p}$, that $j \in \{1, \ldots, p\}$ and $v_j = k$.

The above formal syntax is too formal for convenient use, and it is advantageous to use more suggestive, and compact, notation.

Pi	for	$\texttt{P}_{\sigma,i}$ (that is, for example, P4 for $\texttt{P}_{\sigma,4}$)
Ti	for	$\texttt{T}_{\sigma,i}$
Ti:= e_1	for	$\texttt{ASSIGN}(i, e_1)$
$\rho(e_1, \ldots, e_p)$	for	$\texttt{CALL}(\rho, e_1, \ldots, e_p)$
e_0.i← $\langle e_1, \ldots, e_p \rangle$	for	$\texttt{CHANGE}(e_0, i, \langle e_1, \ldots, e_p \rangle)$
e_0.i.j	for	$\texttt{ACCESS}(e_0, i, j)$.

It will frequently be the case that we want to do a PCHANGE operation such as

$$\text{Pm.i} \leftarrow \langle\ \text{Pm.i.1},\ldots,\text{Pm.i.(j-1)}, e_j, \text{Pm.i.(j+1)},\ldots,\text{Pm.i.p}\ \rangle$$

where "only the *jth* component of Pm.i is changed", we will write this as

$$\text{Pm.i.j} \leftarrow e_j.$$

While this is convenient notation, it is possible to misuse it and write something meaningless. The formal syntax is the real syntax.

Not surprisingly, it will also be convenient to present the data for a class-collection in a more informal manner. We will not attempt to explain these informalities but leave the reader to deduce them from the examples.

5 State Algebras

Given a specification $\Gamma = \langle K, \Sigma, \alpha, \iota, \tau, \xi \rangle$ we want to define the set of many-sorted algebras corresponding to the possible states resulting from executing the methods given for the classes.

We start by defining the collection of Γ-*signatures* corresponding to the data Γ. A Γ-signature $\langle S, \Omega \rangle$ will contain a designated sort **1**, and, if $k \in K$, with $\iota(k) = v_1 \cdots v_n \in (K^*)^*$ and $v_i = k_{i,1} \cdots k_{i,p_i} \in K^*$ then k will contribute $n+4$ elements to the sort set S, namely $V_{k,1}, \ldots, V_{k,n}, S_k, I_k, O_k$, and T_k. Think of $V_{k,i}$ as *the ith sort of instance variables for k*, S_k as *the sort of summands for k*, I_k as *the sort of instances of k*, O_k as *the sort of objects of k*, and T_k as *the sort of temporary variables of k*. These sorts come equipped with operations:

$$\begin{array}{lll} nil_k : \mathbf{1} \to O_k & \tau_k : T_k \to O_k & \kappa_k : I_k \to O_k \\ \mu_k : I_l \to S_k & \iota_{k,i} : V_{k,i} \to S_k, i = 1, \ldots, n & \\ \pi_{k,i,j} : V_{k,i} \to O_{k_{i,j}}, i \in \{1, \ldots, n\}, j \in \{1, \ldots, p_i\}. & & \end{array}$$

In addition, Ω may contain a finite set of constants of sorts O_k and T_k for each $k \in K$.

We can represent this signature pictorially as shown in Figure 1. See Section 7 for pictures of some actual signatures.

A Γ-*state-algebra* (or Γ-algebra), **A**, will be a $\langle S, \Omega \rangle$-algebra, for some Γ-signature $\langle S, \Omega \rangle$, where $\mathbf{A_1}$ is a designated singleton set also denoted **1**, and for each $k \in K$, as above,

> $\mathbf{A}_{O_k} = \mathbf{A}_{I_k} + \mathbf{A_1}$, a coproduct of these sets in **Set**, the category of sets and total functions, with coproduct injections $(\kappa_k)_\mathbf{A}$ and $(nil_k)_\mathbf{A}$.
>
> For each $i \in \{1, \ldots, n\}$, $\mathbf{A}_{V_{k,i}} = \mathbf{A}_{O_{k_{i,1}}} \times \cdots \times \mathbf{A}_{O_{k_{i,p_i}}}$, a product of these sets in **Set**, with product projections $(\pi_{k,i,1})_\mathbf{A}$ through $(\pi_{k,i,p_i})_\mathbf{A}$. If, where $\iota(k) = (v_1) \cdots (v_i) \cdots (v_n)$ we have $v_i = \lambda$, the empty string, then we take $\mathbf{A}_{V_{k,i}} = 1$.

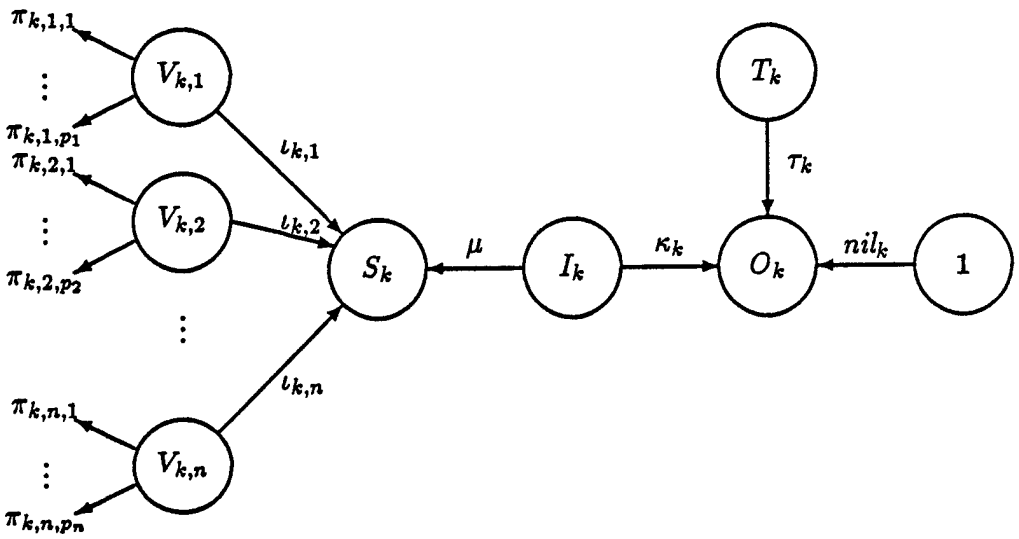


Figure 1: The k-component of a Γ-signature

$\mathbf{A}_{S_k} = \mathbf{A}_{V_{k,1}} + \cdots + \mathbf{A}_{V_{k,n}}$, a coproduct of these sets in **Set**, with coproduct injections $(\iota_{k,1})_\mathbf{A}$ through $(\iota_{k,n})_\mathbf{A}$.

$\mathbf{A}_{T_k}$ will be exactly the set of constants of sort T_k.

We do not put any restrictions on $\mathbf{A}_{I_k}$, $(\tau_k)_A$, or $(\mu_k)_A$, other than that the functions be functions.

Let **A** be a state algebra, then the ideas behind the above definition of state-algebra are as follows:

- An object of class k in **A**, that is, an element of $\mathbf{A}_{O_k}$, is either the nil-object given by nil_k or it is an instance of k, that is, an element of $\mathbf{A}_{I_k}$. This is just what the coproduct says.
- Each instance of an object of class k has a value. In particular, if $x \in \mathbf{A}_{I_k}$ then its value is $(\mu)_A(x) \in \mathbf{A}_{S_k} = \mathbf{A}_{V_{k,1}} + \cdots + \mathbf{A}_{V_{k,n}}$, which is a sum of products of objects.
- Each element y of $\mathbf{A}_{T_k}$ corresponds to a temporary variable with value $(\tau)_A(y)$, an object of class k.

6 Abstract Operational Semantics

Given the class specification $\Gamma = \langle K, \Sigma, \alpha, \iota, \tau, \xi \rangle$, and given $\sigma \in \Sigma$ with $\alpha(\sigma) = \langle j, u, h \rangle$ then a *σ-state algebra (over Γ)* is a Γ-state algebra A whose signature $\langle S, \Omega \rangle$ contains at

least the constant symbols $\mathrm{P}_{\sigma,1},\ldots,P_{\sigma,|u|}$, corresponding to the parameters of σ, together with the constant symbols $T_{\sigma,1},\ldots,T_{\sigma,|\tau(\sigma)|}$, corresponding to the local variables for σ. In practice we are only interested in σ-state algebras with finite carriers, and generated by repeated "applications of body-expressions". From this we can, but won't, show that that the σ-state algebras of interest form a set, Alg_σ, rather than a proper class.

When a k-σ-expression e is applied to a σ-state-algebra A, the result, if any, will be a pair $[\![e]\!]_\sigma(A) = \langle B, b\rangle$ where B is a σ-state-algebra B and b is an element of B_{O_k}. Let RES_σ denote the set of all pairs $\langle A, a\rangle$ such that $A \in Alg_\sigma$ and $a \in A_{O_k}$. Then we can regard $[\![e]\!]_\sigma$ as a partial function,

$$[\![e]\!]_\sigma : Alg_\sigma \rightarrow RES_\sigma.$$

Let $A \in Alg_\sigma$-algebra with signature $\langle S, \Omega\rangle$, then $[\![e]\!]_\sigma(A)$ is defined by the appropriate entry from the following list.

$\mathtt{NIL}_k$: Define $[\![\mathtt{NIL}_k]\!]_\sigma(A) = \langle A, (nil_k)_A\rangle$.

$\mathrm{P}_{\sigma,i}$: Define $[\![\mathrm{P}_{\sigma,i}]\!]_\sigma(A) = \langle A, (\mathrm{P}_{\sigma,i})_A\rangle$.

$\mathrm{T}_{\sigma,i}$: Define $[\![\mathrm{T}_{\sigma,i}]\!]_\sigma(A) = \langle A, (\mathrm{T}_{\sigma,i})_A\rangle$.

$e_1;e_2$: Define $[\![e_1;e_2]\!]_\sigma(A) = [\![e_2]\!]_\sigma(([\![e_1]\!]_\sigma(A))_1)$.

$\mathtt{INST}(e_0, e_1, e_2)$: If e_0 is a z-σ-expression and $[\![e_0]\!]_\sigma(A) = \langle C, c\rangle$, then define

$$[\![\mathtt{INST}(e_0, e_1, e_2)]\!]_\sigma(A) = \begin{cases} [\![e_1]\!]_\sigma(C) & \text{if } c \neq nil_z \\ [\![e_2]\!]_\sigma(C) & \text{if } c = nil_z. \end{cases}$$

$\mathtt{ASSIGN}(i, e_1)$: If $[\![e_1]\!]_\sigma(A) = \langle B, b\rangle$ then $[\![\mathtt{ASSIGN}(i, e_1)]\!]_\sigma(A) = \langle C, b\rangle$ where C is identical to B with the exception that $(\tau_k)_C((T_{\sigma,i})_C) = b$.

$\mathtt{CALL}(\rho, e_1, \ldots, e_p)$: Let $\langle B_1, b_1\rangle = [\![e_1]\!]_\sigma(A)$, and, for $i = 2,\ldots,p$, let $\langle B_i, b_i\rangle = [\![e_i]\!]_\sigma(B_{i-1})$. Then, where $\alpha(\rho) = \langle h, w, k\rangle$ and $\tau(\rho) = u$ extend the signature $\langle S, \Omega\rangle$ by adding constant symbols $P_{\rho,i}, i = 1,\ldots,|w|$ and $T_{\rho,j}, j = 1,\ldots,|u|$, and let B be the extension of B_p such that $(P_{\rho,i})_B = b_i$ for each $i = 1,\ldots,|w|$, and, $(\tau_{u_j})_B(T_{\rho,j}) = nil_{u_j}$, for each $j = 1,\ldots,|u|$. Then, where $\xi(\rho) = e$, $\langle C, c\rangle = [\![e]\!]_\rho(B)$, and D is the $\langle S, \Omega\rangle$-reduct of C, define $[\![\mathtt{CALL}(\rho, e_1, \ldots, e_p)]\!]_\sigma(A) = \langle D, c\rangle$.

$\mathtt{NEW}(k, i)$: Let B be the σ-state-algebra that results from freely adjoining an element x to A_{I_k} and, where $\iota(k) = v_1 \cdots v_n$ and $v_i = v_{i,1} \cdots v_{i,p} \in K^*$, taking $(\mu_k)_B$ to be the extension of $(\mu_k)_A$ taking x to $b = (\iota_{k,i})_A(\langle nil_{v_{i,1}}, \ldots, nil_{i,p}\rangle)$. Then define $[\![\mathtt{NEW}(k, i)]\!]_\sigma(A) = \langle B, b\rangle$.

$\mathtt{CASE}(e_0, e_1, \ldots, e_q)$: If e_0 is a z-σ-expression, where $\alpha(z) = w_1 \cdots w_q$, and $[\![e_0]\!]_\sigma(A) = \langle C, c\rangle$, then define

$$[\![\mathtt{CASE}(e_0, e_1, \ldots, e_q)]\!]_\sigma(A) = \begin{cases} \langle C, (nil_z)_C\rangle & \text{if } c = (nil_z)_C \\ [\![e_i]\!]_\sigma(C) & \text{if there exist x and y such that} \\ & c = (\kappa_z)_C(x) \text{ and } (\mu_z)_C(x) = (\iota_{z,i})_C(y). \end{cases}$$

CHANGE$(e_0, i, e_1, \ldots, e_p)$: Let $\langle B_0, b_0\rangle = [\![e_0]\!]_\sigma(A)$, and, for $i = 1, \ldots, p$ let $\langle B_i, b_i\rangle = [\![e_i]\!]_\sigma(B_{i-1})$. Then $[\![\text{CHANGE}(e_0, i, e_1, \ldots, e_p)]\!]_\sigma(A) = \langle C, b_0\rangle$ where C is identical to B_p except that if b_0 is an instance of k, so $b_0 = (\kappa_k)_{B_p}(x)$ for some $x \in (B_p)_{I_k}$, then $(\mu_k)_C(x) = (\iota_i)_{B_p}(\langle b_1, \ldots, b_p\rangle)$.

ACCESS(e_0, i, j) : Let $[\![e_0]\!]_\sigma(A) = \langle B, b\rangle$, if there exists $x \in A_{I_k}$ such that $b = (\kappa_k)_A(x)$ and $(\mu_k)_A(x) = (\iota_i)_A(\langle a_1, \ldots, a_p\rangle)$, then $[\![\text{PACCESS}(e_0, i, j)]\!]_\sigma(A) = \langle A, a_j\rangle$, otherwise

$$[\![\text{PACCESS}(e_0, i, j)]\!]_\sigma(A) = \langle A, nil_k\rangle.$$

The above presentation of the semantics is a mite informal in that it assumes that, for each k-σ-expression e and Γ-algebra A, $A \subseteq ([\![e]\!](A))_1$ in some sense that makes it meaningful to talk of an object in $x \in A_{O_k}$ as also being an object in $(([\![e]\!](A))_1)_{O_k}$. A more precise treatment would require introducing generalized injective homomorphisms.

7 Examples of Class Specifications

In this section we give a number of examples of class specifications using the sugared version of the syntax. Each specification builds on the ones given before. The specifications are fairly straight forward, but generally represent very inefficient implementations. For example, in the specification of *BOOL* the reader will see that each application of the "constant operation" *true* generates a new object.

Example 1 Here is a specification for the class *BOOL*. The only surprise here may be the operation *null*. This operation is needed because the CASE operation, which distinguishes *true* from *false*, can not used outside of the *BOOL* class. The operation *null* can be used together with the primitive operation INST to give us a general BOOLean conditional usable in methods of any class. The signature diagram for *BOOL* is shown in Figure 2.

```
CLASS BOOL
form
    ι(BOOL) = (λ)(λ)

methods
    true( ):BOOL
        NEW(BOOL, 1).

    false( ):BOOL
        NEW(BOOL, 2).

    and( P1, P2:BOOL):BOOL
        CASE(P1, CASE(P2, true, false), false).
```

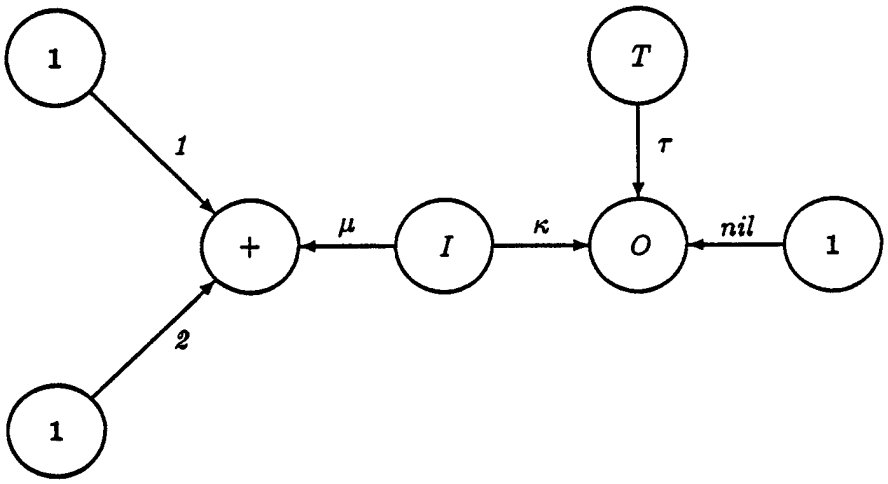


Figure 2: The Signature for BOOL

not(P1*:BOOL):BOOL*
CASE*(*P1*, false, true).*

null(P1*:BOOL):BOOL*
CASE*(* P1*, true,* $\mathtt{NIL}_{BOOL}$*).*

end – *class BOOL*

Example 2 Here is a specification for the class *NAT* of natural numbers. This is an example of a "recursive class" in the sense that *NAT* appears in the specification of the form of *NAT*. In general, a state algebra for *NAT* will contain only a subset of the natural numbers. As examination of the methods will show, "*NAT*s are only produced as needed".

CLASS *NAT*
form
$\iota(NAT) = (\lambda)(NAT)$

methods
zero( ):NAT
NEW*(NAT, 1).*

succ(P1*:NAT):NAT*
(τ*(succ) =* T1*:NAT)*
T1*:=*NEW*(NAT, 2);*T1*.2*$\leftarrow$ $\langle$P1$\rangle$.

pred(P1*:NAT):NAT*
CASE*(* P1*,* $\mathtt{NIL}_{NAT}$*,* P1*.1.1).*

*add(*P*1, *P*2:NAT):NAT*
CASE(P*2*, P*1*, *add(succ(*P*1*), pred(*P*2*))).*

subt(P*1, *P*2:NAT):NAT*
CASE(P*2*, P*1*, *minus(pred(*P*1*), pred(*P*2*))).*

eq(P*1, *P*2:NAT):NAT*
CASE(P*1*, CASE(P*2*, *true*, *false*), CASE(P*2*, *false*, *eq(pred(* P*1*), pred(* P*2*))).*

le(P*1, *P*2:NAT):NAT*
CASE(P*1*, CASE(P*2*, *true*, *false*), *le(pred(* P*1*), pred(* P*2*))).*

end - *class NAT*

Example 3 As our next example we give a specification for the class *INT* of integers. This specification provides an a nice example of encapsulation. From looking at the names of the methods one would expect that an integer z is being represented as a pair consisting of a BOOLean, representing the sign of z, and a NATural number, representing the absolute value of s. But the specification, actually, represents an integer z by a pair, $\langle n, p \rangle$, of natural numbers such that if $n > p$ then $z = m - p$, while if $n \leq p$ then $z = -|p - n|$.

CLASS *INT*
form
$\iota(INT) = (NAT \cdot NAT)$

methods
one( ):INT
$(\tau(one) =$ T*1:INT)*
T:= NEW*(INT, 1)*;T*.1.1.*←*succ(zero)*;T*.1.2*←*zero.*

abs(P*1:INT):NAT*
INST(*null(le(* P*1.1.1*, P*1.1.2*)), *subt(* P*1.1.2*, P*1.1.1*), *subt(* P*1.1.1*, P*1.1.2*)).*

sign(P*1:INT):BOOL*
INST(*null(le(*P*1.1.1*, P*1.1.2*)), *false*, *true).*

sum(P*1*, P*2:INT):INT*
$(\tau(sum) =$ T*1:INT)*
T*1* := NEW*(INT, 1)*;T*1.1.1*←*add(* P*1.1.1*, P*2.1.1*);T*1.1.2*←*add(* P*1.1.2*, P*2.1.2*));T*1*

neg(P*1:INT):INT*
$(\tau(neg) =$ T*1:INT)*
T*1* := NEW*(INT, 1)*;T*1.1.1*←P*1.1.2*;T*1.1.2*←P*1.1.1*;T*1.*

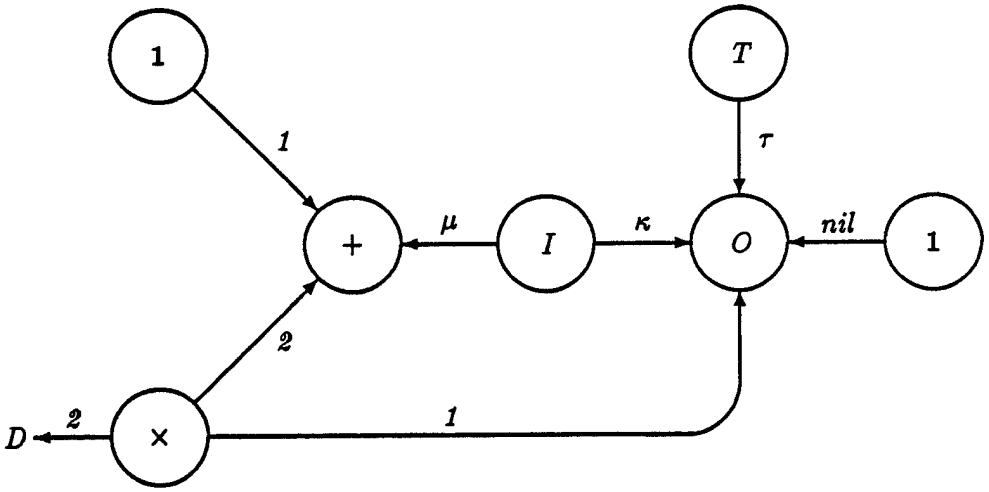


Figure 3: The Signature for STACK(D)

eqint(P*1*, P*2:INT):BOOL*
*eq(add(*P*1.1.1, tP2.1.2), add(*P*1.1.2,* P*2.1.2)).*
end - *class INT*

Example 4 Here is the "classic example" of a data type specification, *STACK(D)*, here presented as a *generic class*, that is, *D* is a *formal parameter* that may be "passed" any *actual parameter* such as *BOOL*, *NAT*, or *INT*. In this paper we will not go into the mathematics of "how parameters are passed" - essentially we use the familiar pushout construction from the theory of data types. Informally, all we have to do is "rewrite" the specification with *D* replaced by the name of the desired actual parameter. The signature diagram for *STACK(D)*) is shown in Figure 3.

CLASS *STACK(D) = STACK-OF-D*
form
ι(STACK(D)) = (λ)(STACK(D)·D)

methods
pop(P*1:STACK(D)):D*
(τ(pop) = T*1:D)*
CASE(P*1*, T*1*:=NIL$_D$, (T*1*:=P*1.2.2*;CASE(P*1.2.1*,
P*1.1*← ⟨ ⟩,
P*1.2*← ⟨(P1.2.1).2.1,(P1.2.1).2.2)⟩;
T*1.*

push(P*1:STACK(D),* P*2:D):D*

```
        (τ(push) = T1:STACK(D))
        T1:=NEW(STACK(D), 2);
        CASE(P1,
            T1.1← ⟨ ⟩,
            T1.2 ← ⟨P1.2.1, P1.2.2⟩);
        P1.2← ⟨T1, P2 ⟩;
        P2.

    make( ):STACK(D)
        (τ(make) = T1:STACK(D))
        T1:=NEW(STACK(D), 1).

    empty?( P1:STACK(D)):BOOL
        CASE( P1, true, false).
end - class STACK(D)
```

Example 5 Our next specification is for DOUBLE-LINKS-OF-D, or *2LINK(D)* – the "links" used to make types such as doubly-linked lists, and used below to specify *FINITE-SETS-OF-D*. One can think of an instance x of a *2LINK(D)* object as having a value which is a triple $\langle lft, rgt, val \rangle$ where lft and rgt are *2LINK(D)* objects and val is an object of class D. Informally, we think of lft as being to the left of x, and rgt as being to the right of x. These leads naturally to the the idea of a doubly-linked-list – a chain of 2LINK(D)s with *nil*'s at the two ends, but *2LINK(D)* can also be used to construct many other structures.

This class is again an example of a class that is both recursive and generic. In contrast to our other examples, this class is quite ill-behaved in that we can have complex structures of links with very complex aliasing. This complexity is largely hidden in this specification in as much as the choice of names for the methods only suggest the doubly-linked list application.

```
CLASS 2LINK(D) = DOUBLE-LINKS-OF-D
form
    ι(2LINK(D)) = (2LINK(D)·2LINK(D)·D)

methods
    left( P1:2LINK(D) ):2LINK(D)
        INST(P1.1.1, P1.1.1, P1 ).

    right( P1:2LINK(D) ):2LINK(D)
        INST(P1.1.2, P1.1.2, P1 ).

    addleft( P1:2LINK(D), P2:D ):2LINK(D)
        (τ(addleft) = T1:LINK(D))
        T1:=NEW(2LINK(D), 1);
        INST( P1,
```

INST(P1.1.1,
T1.1← ⟨P1.1.1, P1, P2⟩;(P1.1.1).1.2←T1;P1.1.1←T1,
T1.1← ⟨$\mathtt{NIL}_{2LINK(D)}$, P1, P2⟩;P1.1.1←T1);
T1.1← ⟨$\mathtt{NIL}_{2LINK(D)}$, $\mathtt{NIL}_{2LINK(D)}$, P2⟩).

addright(P1:2LINK(D), P2:D):2LINK(D)
left to the reader

write(P1:2LINK(D), P2:D):D
P1.1.3←P2.

leftend?(P1:2LINK(D)):BOOL
INST(P1, $\mathtt{NIL}_{BOOL}$, INST(P1.1.1, *false, true*)).

rightend?(P1:2LINK(D)):BOOL
left to the reader

drop(P1:2LINK(D)):2LINK(D)
INST(*null(leftend?(* P1 *))*,
INST(*null(rightend?(* P1*))*, $\mathtt{NIL}_{2LINK(D)}$, *(right(*P1*)).1.2*←$\mathtt{NIL}_{2LINK(D)}$),
INST(*null(rightend?(* P1*))*,
*(left(*P1*)).1.2*←$\mathtt{NIL}_{2LINK(D)}$,
*(right(*P1*)).1.1*←*left(*P1*);(left(*P1*)).1.2*←*right(*P1).

Finally, assuming D has an "equality method", *eqD(* P1. P2:D *):BOOL*, then

isin?(P1:D, P2:2LINK(D) *):BOOL*
*(*τ*(isin?)* = T1:BOOL)
INST(P2,
INST(*null(eqD(* P2.1.3, P1*))*, *true*, *isin?(* P1, P2.1.2*)*),
false).

end – *class 2LINK(D)*

Example 6 As our final example we give a specification for the generic class *FINITE-SETS-OF-D*. The specification makes use of the class *2LINK(D)* but the *2LINK(D)*s generated by the methods in *SET(D)* are encapsulated in the sense that there is no way "to get at them" except through the methods of *SET(D)*. Note that this generic specification makes use of the *isin?* method of *2LINK(D)* and thus requires that D has an "equality method" *eqD*. The idea behind this specification is that we can represent a set s as a "string" of its elements, and that we can represent the string as a chain of *2LINK(D)*. In the actual specification the form of *SET-OF-D* is given as a triple, $\langle L_1, L_2, L_3 \rangle$ of *2LINK(D)* objects. Inspection of the methods should show that, in a string s representing a set S, L_1 marks

the beginning of s, L_3 marks the end of s, and L_2 is used, when necessary, to traverse s, but is always returned to the beginning of s at the end of a method.

```
CLASS SET(D) = FINITE-SETS-OF-D
form
    ι(SET(D)) = ( 2LINK(D)·2LINK(D)·2LINK(D) )

methods
    make( ):SET(D)
        (τ(make) = T1:SET(D))
        T1:=NEW(SET)D), 1).

    elemof?( P1:D, P2:SET(D) ):BOOL
        (τ(elemof?) = T1:BOOL)
        T1:= isin?( P1, P2.1.2);P2.1.2←P2.1.1;T1.

    addelem( P1:D, P2:SET(D) ):SET(D)
        INST( null(elemof?( P1, P2)),
            P2,
            P2← ⟨ P2.1.1, P2.1.2, (addright( P2.1.3, P1);right(P2.1.3)) ⟩ ).

    delelem( P1:D, P2:SET(D) ):SET(D)
        (τ(delelem) = T1:2LINK(D))
        INST( null(isin?( P1, P2.1.2)),
            T1:=drop( P1, P2.1.2);INST( null(leftend?(P2.1.2)),
                P2← ⟨T1, T1, T1⟩,
                P.1.2←P1.1 ),
            P2.1.2←P2.1.1 ).
end - class SET(D)
```

8 Looking Forward

This paper has concentrated on giving a description of a particular language for data driven design and on showing some simple examples of what can be done with it. But the real reason for developing the language was, and is, to use it as a well defined framework in which to investigate various aspects of data driven design. I am not ready, at present, to make any major pronouncements on data driven design, but the following remarks, and questions, may be of interest.

While you may not be completely happy with the way I have worked out the examples in Section 7 you will probably agree that most, if not all, of them are correct. But what does this mean? Intuitively, it means that the defined classes have the external behavior that we expect. What is "external behavior"? I think that, loosely speaking, external behavior is

what we can observe by doing experiments consisting applying expressions built up using the "public" operations INST, NIL, ASSIGN, CALL and ;. This is trickier than it might sound since essentially all we can observe as the result of an experiment is whether or not the result is a *nil* object. The idea is that the "experiments" should provide a way to identify appropriate states and/or objects so that the resulting congruence classes correspond to the elements of the desired abstract type. It seems fairly easy to make this precise in a manner that will work for at least *BOOL*, *NAT*, *STACK-OF-D*, and *FINITE-SET-OF-D*. However, we can take the specification given for *FINITE-SET-OF-D*, rename it *NON-REPEATING-STRING-OF-D*, and informally interpret the objects as strings of elements of D in which no element is repeated. This is fine intuitively, but the above notion of external behavior is too strong as it identifies strings that are not the same under this new interpretation. However, interpretation not withstanding, any application of *NON-REPEATING-STRING-OF-D* we can replace it by *FINITE-SET-OF-D* and never know the difference. Still, it would appear that the meaning of a class involves intention as well as extension. At the very least it means that we can not necessarily grasp the intention behind a class specification just from the formal specification.

An aspect of object-oriented programming that receives a great deal of attention is the notion of "inheritance". This is a "concept" with many definitions, some of which seem to be incompatible. The version I want to address is roughly the intersection of the versions found in [3] and [2], to quote from [3]:

> "Inheritance is a technique that allows new classes to be built on top of older, less specialized classes rather than written from scratch. The new class is the *subclass*; the old one is the *superclass*. The subclass *inherits* the instance variables and methods of the superclass. The subclass can add new instance variables and methods of its own."

To put this into the framework of our language we need only replace the first occurrence of the phase "instance variables" by the phrase "form ι", and replace the phase "add new instance variables" by a phrase describing some suitable notion of extending the form. The question of what is suitable notion can wait until another day, what is important is that this is an implementation concept in the sense that the new class, k', is defined starting from the specification $\langle \alpha(k), \iota(k), \xi(k) \rangle$ of the old class k, rather being defined from the external behavior of k.

The fact that "inheritance" works at the implementation level results in some confusions at the interpretation level. We can easily take the class *FINITE-SET-OF-D* and add a method for a *pick operation* which, when applied to a "set" s, returns "the oldest element of s" – we just return the right-most element of D in the "string" representing s. I claim that the resulting class is constructed in accordance with the directions given in the above quote, and is thus technically a "subclass" of *FINITE-SET-OF-D*. Now it seems wrong, from a mathematical point of view, to say that the result is a specialization of the mathematical concept of finite sets of elements of D. However, there is no problem with viewing the resulting class is a specialization, or extension, of the class *NON-REPEATING-STRING-*

OF-D. This suggests that there are important semantic elements in inheritance that need further investigation.

It is worthwhile considering if there are other ways to "reuse code", that do not lead to such semantic problems. Certainly we can extend a class by adding methods that are defined in terms of existing methods by means analogous to the construction of derived operators in universal algebra. But can we do better than this? I think we can.

In Section 7, our examples are developed sequentially. That is, there are no mutually recursive specifications. However, it is certainly possible to write such specifications in our language. For example the definition of the class *STACK-OF-D* could be broken into two separate class definitions $S = STACK$ and $N = NON\text{-}EMPTY\text{-}STACK$ where, $\iota(S) = (\,)(N)$ and $\iota(N) = (S \cdot D)$. Are such specifications needed or useful?

I'll present some answers, and more questions, in my talk.

References

[1] Goldberg, Adele and Robson, David, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass., 1983.

[2] Meyer, Bertrand *Object-Oriented Software Construction*, Prentice-Hall International Series in Computer Science (C.A.R. Hoare, Series Editor) Prentice-Hall, New York, 1988.

[3] Micallef, J., "Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages," *JOOP*, April/May 1988.

[4] Wolczko, Mario, *Semantics of Object-Oriented Languages* Doctoral dissertation: Department of Computer Science, University of Manchester, Technical Report Series UMCS–88–6–1, 1988.

[5] Wagner, Eric G. "Categorical Semantics, or Extending Data Types to Include Memory," *Recent Trends in Data Type Specification – Proceedings of the3rd Workshop on Theory and Applications of Abstract Data Types* (H.-J. Kreowski, Ed.) Informatik-Fachberichte 116, Springer-Verlag (1985).

[6] Wagner, Eric G., "On Declarations", The International Workshop on Categorical Methods in Computer Science (Berlin, Sept. 88)

[7] Wagner, Eric G., "All recursive types defined using products and sums can be implemented using pointers," To appear in the Proceedings of The Workshop on Algebraic Logic and Universal Algebra in Computer Science, Iowa State University, 31 May-5 June, 1988.

[8] Wagner, Eric G., "Semantics of Block Structured Languages with Pointers," Proceedings of 3rd workshop on Mathematical Foundations of Programming Language Semantics, LNCS 298, Springer-Verlag (1988) pp 57-84.

[9] Thatcher, James, W., Wagner, Eric G., Wright, Jesse B., "Data type specification: Parameterization and the power of specification techniques," *ACT Trans. Progrramm. Lang. and Systems* 4 (1982) pp 711-732.

[10] Goguen, Joseph A., Thatcher, James W., Wagner, Eric G., "An initial algebra approach to the specification, correctness, and implementation of abstract data types," *Current Trends in Programming Methodology, IV: Data Structuring* (R. Yeh, ed.) Prentice Hall, New Jersey (1977).

[11] Wagner, Eric G., Selker, Ted, Rutledge, Joseph, "Algebraic Data Types and Object Oriented Programming," Sixth Workshop on Specification of Abstract Data Types, Berlin, West Germany, Aug 29- Sep 2, 1988. Paper, with possibly a different title, in preparation.

Modelling the Software Process

Charles Rattray
Department of Computing Science
University of Stirling
STIRLING, Scotland FK9 4LA
e_mail: cr@uk.ac.stir.cs

1 Introduction

Software Engineering aims to improve our ability to develop and maintain provably correct, adaptable, and efficient software. Initial attempts to provide this improvement were based on the software development process known as the software life-cycle.

Indeed, software engineering has matured to the point where some of its fundamental premises should be re-examined. In particular, the traditional view of the "software life-cycle" has been recognised as inadequate when considered with automated environments based on "rapid prototyping" or "knowledged-based development" or the transformation paradigm [Wil86a,ICS87,Hen89,Rat88b,Rat88c,Rat89b].

Further, in any traditional engineering discipline, re-usability (of components, designs, manufacturing processes, ...) is of fundamental importance. In software engineering, we are just beginning to understand this idea, but before we can incorporate it into our software practices we must understand the process of software development. To do this we must understand both the constituents of the development process, and the total process itself. The common approaches to software re-usability are well-illustrated by work involving a re-usability of system components (in the form of libraries, etc [Hor84]), abstract data types [Emb87,Gog86], and specification [Gau88]. Software re-use can be identified with the productive re-use of software design and development in the planning, construction, and verification of software systems; a knowledge-based model for this form of software process re-use is described in Rattray et al [Rat89b].

A number of approaches to the study of the software process [Wil86a,ICS87,Hen89] have been suggested. Typical is that of Osterweil [Ost87] where the view is put forward that software processes can be described by "programming" them in much the same way as computer applications are programmed. A criticism of this by Lehman [Leh87] is that a process program is essentially procedural and only has merit if the problem domain is known and well-understood, if the strategies and algorithms for achieving the desired goals are known, and if the managerial and administrative practices are clearly defined.

Similar criticisms can be levelled at other attempts to describe software processes and so the need to develop (mathematical) models or meta-models of the software process has become essential. Dowson [Dow86] gives reasonable definitions for "software process",

"software process meta-model" and "software process model"; Wileden [Wil86b], for instance, provides a possible meta-model which agrees with Dowson's definitions. None of the models available seem appropriate. They lack precision, comprehensiveness, consistency, and an adequate theoretical basis, and most importantly the notion of variable structure [Zei89]. These difficulties are all overcome by a formal (meta-)model, evolutionary and hierarchical in nature and based on a re-interpretation of elementary categorical algebra, developed by Rattray [Rat88b] and Rattray and Price [Rat89a]. This model provides a suitable framework within which to consider software process development and re-use.

Within this evolving hierarchical framework, systems have an internal organisation consisting of components with interrelations; their organisation is maintained in time even though their components are changing; their components are divided on levels corresponding to the increasing complexity of their own organisation. The state of a system at any given time is modelled by a category, the state transition by a functor, a complex component by the (inductive) limit of a pattern of linked components. Categorical constructions describe the stepwise formation of a system, by means of operations: absorption of external components from the environment, destruction of some components, formation of new complex components. By defining the notion of a mapping (morphism), compatible with the evolving hierarchical structure, between frameworks it is possible to compare software development processes.

In many situations, software maintenance for example, our knowledge of the software product may not be complete (lack of documentation, change of personnel, etc). Information to and from the software system may then be supposed to be conveyed through limited parts of it called "actors", dynamically interacting with the system. Each actor has only partial information of the system. For each actor, we can construct a category, its "field of vision", which contains the fragments of the software system available to it; these fields are connected via "communication" functors. From this, we can deduce a representation of the totality of fragments attainable through the actors. For the outside observer the actors' view need not be a faithful representation of the actual software system and the difference is measured by a "distortion" functor. The actual system may be subject to modifications outside the scope of the actors. The difference between the software system as "anticipated" by the actors and the system after external modification can be measured by a comparison functor; the measurement represented by the comparison functor is available at the level of the actors and indicates how to reduce the difference.

In this paper, we review the framework, which has its origin in the biological sciences [Ehr85,Ehr86], as a vehicle in which to consider models of the software process. Using the kinds of measurements mentioned above we illustrate how it is possible to devise construction strategies for building complex systems or understanding existing software products. As an application of the evolutionary hierarchical framework, we indicate how the model may be helpful in understanding software re-usability by considering this from the point of view of re-usability of software processes [Rat88a]. The same framework is being developed to provide the underlying model for the design of a practical experimental program design environment [Rat89a].

2 Elements of a Meta-Model

"A system is a set of units with relationships among them" [Ber56]. The notion of a category matches well with this description whereby the objects of the category model the system units and the arrows of the category model the system relationships. Graphical descriptions of system models from which the software is generated are equally common. Typically, SADT[1](Structured Analysis and Design Technique [Mar87]) uses hierarchies of directed graphs the nodes of which have a particular form. Each node is an abstraction (complex component) the internal organisation of which is described by a directed graph.

2.1 Complex Components

A *pattern* is a graph morphism $P : G \rightarrow K$ from a graph G to a graph K. Graph G prescribes the *shape* of the diagram and may be viewed as a sketch (a *prototype*) of a system's structure, ie pattern P defines a *pattern of linked (related) objects in* K.

Suppose $\mathbf{K}$ is a category of such objects and P is a pattern of objects in $\mathbf{K}$, whose prototype is G. The object P_i, indexed by the node i of the prototype is called a *component of* $\mathbf{K}$; the image $P(a)$ of an arrow a of the prototype is a *specific link* between components. Such a specific link indicates a relationship between the objects, eg a form of module coupling, a data flow, file transfer, network link, a dependency relation, ...

The key notion in developing complex components of any software system lies in the need to ensure that system changes have only a localised effect. Essentially, the complex component has a certain external behaviour, determined by its internal organisation and function which is unknown and unavailable to the environment of the component. Thus, internal changes to the component which preserve the external behaviour will have no adverse effects on the environment.This localisation property is known as "information hiding" [Par72].

The idea of stepping back from the detailed internal behaviour of a component so that our understanding of the component is determined by its external behaviour is called *abstraction.* An excellent definition [Weg76], due to Wegner, says

> An abstraction of an object is a characterisation of the object by a subset of its attributes .... If the attribute subset captures the "essential" attributes of the object, then the user need not be concerned with the object itself but only with the abstract attributes.

A *collective link* of the pattern P to the object C of the category is a family of arrows f_i, indexed by the nodes of the prototype G, where f_i is an arrow from the component P_i to C, which satisfies the *compatibility condition*: if a is an arrow from i to j in the

[1]SADT is a trademark of SofTech, Inc.

prototype, then f_i is the composition of $P(a)$ and f_i, ie

$$f_i = (P_i \xrightarrow{P(a)} P_j \xrightarrow{f_j} C)$$

An *inductive limit* of pattern P is an object C' of the category **K** such that, for any object C, the arrows from C' to C are in one-one correspondence with the collective links from P to C. The unique arrow f associated with the collective link (f_i) is said to *bind* the f_is.

Thus, the limit object binds together the component objects according to their internal organisation determined by the corresponding prototype. The component P_i of the pattern P is called a component object of the limit C'. The properties of an object depend on the number and nature of the arrows which link it to other objects of the category **K**. It is natural to compare the properties of the complex object C' with those of its components.

The category **K** models the environment of the pattern P. Modification of the environment category may take various forms. For instance, enlarging **K** to **L** or blurring the distinction between two **K** objects in **L** may make retaining the limits difficult. It may be that a pattern P cannot be bound to a limit in **K** but can be forced to have a limit in an extended environment **L**.

A modification to the environment will be modelled by a functor F from **K** to a "new" category **L**. The *image pattern* of P by F is the pattern Q of linked objects in **L** defined by the composition of P and F. Functor F preserves collective links and cones but not necessarily limits. Changing the environment **K** may change the behaviour of a pattern of linked objects. This suggests the possibility of changing the environment purposely so that complex objects binding given patterns of linked objects may be formed. To achieve this a functor F from **K** to the "smallest possible" category **L** containing **K** must be constructed such that the image of each pattern P admits a limit in **L**, and the image of each given cone is a limit-cone.

2.2 Hierarchical Systems

A *hierarchical system* is a category **K** in which the objects are distributed on levels $(0, 1, \ldots, p)$, such that each object of level $n+1$ $(n < p)$ is the limit in **K** of a pattern P of linked objects on level n.

In such a hierarchical system the system components are associated with levels corresponding to increasing complexity of their internal organisation. Any object at level $n+1$ is the limit of a pattern of linked objects at level n but it may form part of a pattern of linked objects whose limit is at level $n+2$. A functor between hierarchical systems *preserves hierarchies* if it does not raise the level of an object and, indeed, any two hierarchical systems may be compared from some particular level upwards.

2.3 Evolution of a System

Software changes with time (in its development phase and during its use); new components are formed, either added from an external source or by construction from simpler components; old components may be re-organised or discarded.

To model this situation the state of the system at "time" t is represented by a category $\mathbf{K}_t$, and its state transition is determined by a functor from $\mathbf{K}_t$ to $\mathbf{K}_{t*}$, its state at a later time $t*$ (there is no requirement for the objects and arrows at t and $t*$ to be the same). A component is "new" at time t if it has no earlier state.

An evolutionary (hierarchical) system K is just a functor from a subcategory $\mathbf{T}$ of the category of time, **Time**, to a category of categories.

To compare software process models within this system we need to define a morphism between evolutionary systems, ie. to compare states of the systems at corresponding times.

Let K be an evolutionary system on $\mathbf{T}$, and $\mathcal{L}$ an evolutionary system on $\mathbf{U}$ (a subcategory of **Time**). A morphism from K to $\mathcal{L}$ consists of a functor φ from $\mathbf{T}$ to $\mathbf{U}$ and a natural transformation from K to the composite of φ and $\mathcal{L}$. This leads to the category of evolutionary systems.

3 Conclusions

A meta-model for the software process has been outlined. This is based on some elementary properties of categorical algebra. The meta-model provides the framework within which to discuss software process models, to compare them, and perhaps to develop new ones. The same framework can lead quite naturally to the design of a knowledge-based software design environment which promotes the notion of software process re-usability.

References

[Ber56] L von Bertalanffy: *Les Problèmes de la Vie*, Gallimard, Paris, 1956.

[Dow86] M Dowson: "The Structure of the Software Process", in [Wil86a].

[Ehr85] A C Ehresmann, J-P Vanbremeersch: "Systemes Hierarchiques Evolutifs: une modelisation des systemes vivants", Prepublication No 1, Universite de Picardie, 1985.

[Ehr86] A C Ehresmann, J-P Vanbremeersch: "Systemes Hierarchiques Evolutifs: modele d'evolution d'un systeme ouvert par interaction avec des agent", Prepublication No 2, Universite de Picardie, 1986.

[Emb87] D W Embley, S N Woodfield: "A Knowledge Structure for Re- Using Abstract Data Types", in [ICS87].

[Gau88] M C Gaudel, T Moineau: "A Theory of Software Re-Usability", ESOP '88, LNCS 300, Springer-Verlag, 1988.

[Gog86] J Goguen: "Re-Using and Interconnecting Software Components", IEEE Computer, **19**, 2, 1986.

[Hen89] *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (editor: P B Henderson), SIGPLAN Notices, **24**, Feb 1989.

[Hor84] E Horowitz, J B Manson: "An Expansive View of Re-Usable Software", IEEE Trans on Software Engineering, **SE-10**, 5, 1984.

[ICS87] *Proceedings of the 9th International Conference on Software Engineering*, Monterey, California, IEEE Computer Society Press, 1987.

[Leh87] M Lehman: "Process Models, Process Programs, Programming Support", in [ICS87].

[Mar87] D A Marca, C L McGowan: *SADT: Structured Analysis and Design Technique*, McGraw-Hill, 1987.

[Ost87] L Osterweil: "Software Processes are Software too", in [ICS87].

[Par72] D L Parnas: On the criteria to be used in decomposing systems into modules, Communications of the ACM, **15**, 12, December 1972.

[Rat88a] C Rattray: "Evolutionary Hierarchical Systems: a Categorical Model for the Software Life-Cycle", IMA Conference on Mathematical Structures for Software Engineering, Manchester, July 13-15, 1988.

[Rat88b] C Rattray, J McInnes, A Reeves, M Thomas: "Knowledge-Based Software Production: from Specification to Program", Alvey Conference, Swansea, 1988.

[Rat88c] C Rattray, J McInnes: "Software Re-Usability in a Knowledge- Based Environment", 3rd Annual Knowledge-Based Assistant Conference, Rome Air Development Center, Utica, 1988.

[Rat89a] C Rattray, D Price: "Sketching an Evolutionary Hierarchical Framework for Knowledge-Based Systems Design", EUROCAST '89, Las Palmas, Feb 26 - Mar 3, 1989.

[Rat89b] C Rattray, J McInnes, A Reeves, M Thomas: "A Knowledge-Based Model for Software Re-Usability", in **Artificial Intelligence and Software Engineering** (editor: D Partridge), Ablex Publ Co, 1989.

[Weg76] P Wegner: Programming languages – the first 25 years, IEEE Transactions on Computers, **C-25**, 12, December 1976.

[Wil86a] *Proceedings of the International Workshop on the Software Process an Software Environments*, (editors: J C Wileden, M Dowson), ACM Software Engineering Notes, **11**, 4, Aug 1986.

[Wil86b] J C Wileden: "This is IT: a Meta-Model of the Software Process", in [Wil86a].

[Zei89] B Zeigler, H Prahofer: " Systems Theory Challenges in the Simulation of Variable Structure Systems", EUROCAST '89, Las Palmas, Feb 26 - Mar 3, 1989.

Path Grammars

Charles Wells
Department of Mathematics
Case Western Reserve University
Cleveland, OH 44106
USA

1 Introduction

This note introduces the concept of **path grammar**, which allows the specification of paths in a directed graph by a method generalizing the ordinary concept of grammar for strings in an alphabet. The concept of derivation and the special notion of context-free path grammar are defined. A pumping lemma for context-free path languages is stated.

2 Graphs and 2-graphs

By **graph** we mean a directed graph; we allow loops and we allow more than one arrow between the same pair of nodes. A graph generates a free category with the universal property that every graph homomorphism to the underlying graph of a category lifts to a functor. A node n of a graph is a **source** if there is a path from n to each other node of the graph, and a **sink** if there is a path *to* n from each other node in the graph.

A **2-graph** is a graph with possibly some 2-cells. A 2-cell may be thought of as an arrow between paths. Precisely, a 2-cell has a source and a target, each of which is a path in the graph with the *same beginning and ending nodes*. The source and target of the 2-cell may be the same and there may be more than one 2-cell between the same two paths.

A **2-category** is a category $\mathcal{C}$ with, for each pair of objects A, B, a category structure on $\mathrm{Hom}(A, B)$ satisfying certain requirements spelled out, for example, in reference [K]. The arrows of the category $\mathrm{Hom}(A, B)$ are called 2-cells. A 2-graph generates a free 2-category with universal property analogous to that of the free category generated by a graph.

3 Path grammars

A path in a graph is a generalization of a string in an alphabet: it is a generalization because one can describe the characters in the alphabet as loops in a one-node graph. In general, one does not get arbitrary strings of arrows in the graph: they must compose head to tail. This suggests the possibility of describing the well formed programs of a programming language as paths in a graph in which the nodes are the types and the arrows are the operations. The composition operation is automatically typechecked in such a description.

3.1 Grammars The concept of 2-graph allows the possibility of building a theory of grammars for paths in a graph which is analogous to and incorporates the ordinary concept of grammar or production system.

Definition 3.1 A **path grammar** $\mathcal{G} = (\mathbf{G}, \mathbf{V}, \mathbf{T}, S)$ consists of a 2-graph $\mathbf{G}$ whose arrows are the union of two disjoint sets $\mathbf{V}$ (the **variables**) and $\mathbf{T}$ (the **terminals**), together with a distinguished arrow S in $\mathbf{V}$. The 2-cells are called **productions**.

Definition 3.2 The grammar $(\mathbf{G}, \mathbf{V}, \mathbf{T}, S)$ is **context-free** if the beginning of every production is a path of length 1.

3.2 Derivations A context-free grammar in the usual sense comes with the concept of a derivation tree of a string. The author is unaware of generalizations of this concept to larger classes of grammars. However, work of A. J. Power [P] leads to a natural general idea of derivation.

Definition 3.3 A **pasting scheme** is a planar graph $\mathbf{D}$ with the following properties:

P.1 $\mathbf{D}$ has a source and a sink.

P.2 For every interior face F of $\mathbf{D}$, there are distinct vertices $s(F)$ and $t(F)$ and directed paths $\sigma(F)$ and $\tau(F)$ from $s(F)$ to $t(F)$ such that the boundary of F is $\sigma(F)[\tau(F)]^R$.

$\mathbf{D}$ is a **context-free** pasting scheme if for each face F, the path $\sigma(F)$ required by P.2 is of length 1.

A pasting scheme $\mathbf{D}$ has a canonical 2-graph structure whose underlying graph is $\mathbf{D}$ and which has one 2-cell $\alpha(F) : \sigma(F) \to \tau(F)$ for each face F of $\mathbf{D}$.

Definition 3.4 Let $\mathcal{G} = (\mathbf{G}, \mathbf{V}, \mathbf{T}, S)$ be a path grammar. A **derivation** consists of

D.1 A pasting scheme $\mathbf{D}$.

D.2 A 2-graph homomorphism h from the canonical 2-graph structure on the scheme $\mathbf{D}$ to $\mathbf{G}$.

If the external boundary of $\mathbf{D}$ is $\sigma\tau^R$, where σ and τ are paths from the source to the sink of $\mathbf{D}$, and σ and τ are labeled via h by paths w and x respectively, then $\mathbf{D}$ is said to be a **derivation of x from w.**

It follows from Theorem 3.3 of Power [P] that the 2-cells in a derivation compose to a unique 2-cell in the free category generated by $\mathbf{G}$. In the case of an ordinary context-free grammar (so $\mathbf{G}$ has one node) a derivation is equivalent to what is called a derivation tree in [HU].

4 The language of a grammar

Theorem 4.5 *Let $\mathcal{G} = (\mathbf{G}, \mathbf{V}, \mathbf{T}, S)$ be a grammar. The* **language** *$L(\mathcal{G})$ of $\mathcal{G}$ is the set of paths w in $\mathbf{G}$ with the properties:*

L.1 *All of the arrows in w are in $\mathbf{T}$.*

L.2 *There is a derivation of w from S.*

A set L of paths in a graph is **context free** if the graph underlies the 2-graph $\mathbf{G}$ of a finite context-free path grammar $\mathcal{G}$ and L is the language of $\mathcal{G}$.

When a grammar is applied to the specification of programs in a functional programming language, a particular choice of initial arrow S produces the set of all programs with specific input and output types.

4.1 A pumping lemma

The following theorem is a generalization of the pumping lemma for ordinary context free grammars and is proved in the same way.

Theorem 4.6 *Let L be a context-free set of paths in a graph G. Then there is an integer n for which, if z is a path in L of length greater than n, then there is a composable sequence $\langle u, v, w, x, y\rangle$ of paths in G with the following properties:*

PL.1 *The composite vwx has length $< n$.*

PL.2 *Both v and x are loops.*

PL.3 *Either v or x is nonempty.*

PL.4 $z = uvwxy$.

PL.5 *For every nonnegative integer m, $uv^m wx^m y \in L$.*

5 Remarks

Context-free grammars have long been used as a first cut in defining programming languages. These do not completely define the language because of additional context-sensitive restrictions such as type checking and bound checking. Type checking, but not bound-checking, is handled automatically by the use of context-free path grammars.

By adding equations on the paths and requirements on the nodes of a path-grammar which force them to be limits (as in the theory of sketches, [WB] and [W]) it should be possible to handle bound-checking as well. This is the subject of current joint work with A. J. Power.

6 References

[HU] J. E. Hopcroft and J. D. Ullman, **Introduction to Automata Theory, Languages and Computation.** Addison-Wesley, 1979.

[K] G. M. Kelly, **Basic Concepts of Enriched Category Theory.** Cambridge University Press, 1982.

[P] A. J. Power, *A 2-categorical pasting theorem.* J. Alg., to appear.

[W] C. Wells, *A generalization of the concept of sketch.* To appear in Theoretical Computer Science.

[WB] C. Wells and M. Barr, *The formal description of data types using sketches*, in M. Main *et al*, eds., **Mathematical Foundations of Programming Language Semantics.** Lecture Notes in Mathematics 298, Springer-Verlag, Berlin, Heidelberg, New York, 1988.

Author's e-mail address: WELLS@CWRU.BITNET

ENRICHED CATEGORIES AND THE FLOYD-WARSHALL CONNECTION

Vaughan Pratt
Computer Science Dept.
Stanford University
April, 1989

Abstract

We give a correspondence between enriched categories and the Gauss-Kleene-Floyd-Warshall connection familiar to computer scientists. This correspondence shows this generalization of categories to be a close cousin to the generalization of transitive closure algorithms. Via this connection we may bring categorical and 2-categorical constructions into an active but algebraically impoverished arena presently served only by semiring constructions. We illustrate these techniques by applying them to Birkoff's poset arithmetic, interpretable as an algebra of "true concurrency."

The Floyd-Warshall algorithm for generalized transitive closure [AHU74] is the code fragment

$$\textbf{for } v \textbf{ do } \textbf{for } u, w \textbf{ do } \delta_{uw} \mathrel{+}= \delta_{uv} \cdot \delta_{vw}.$$

Here δ_{uv} denotes an entry in a matrix δ, or equivalently a label on the edge from vertex u to vertex v in a graph. When the matrix entries are truth values 0 or 1, with $+$ and $\cdot$ interpreted respectively as $\vee$ and $\wedge$, we have Warshall's algorithm for computing the transitive closure δ^+ of δ, such that $\delta^+_{uv} = 1$ just when there exists a path in δ from u to v. When the entries are nonnegative reals, with $+$ as min and $\cdot$ as addition, we have Floyd's algorithm for computing all shortest paths in a graph: δ^+_{uv} is the minimum, over all paths from u to v in δ, of the sum of the edges of each path.

Other instances of this algorithm include Kleene's algorithm for translating finite automata into regular expressions, and Gauss's algorithm for inverting a matrix, in each case with an appropriate choice of semiring.

Not only are these algorithms the same up to interpretation of the data, but so are their correctness proofs. This begs for a unifying framework, which is found in the notion of semiring. A semiring is a structure differing from a ring principally in that its additive component is not a group but merely a monoid, see AHU [AHU74] for a more formal treatment.

Other matrix problems and algorithms besides Floyd-Warshall, such as matrix multiplication and the various recursive divide-and-conquer approaches to closure, also lend themselves to this abstraction.

This abstraction supports mainly vertex-preserving operations on such graphs. Typical operations are, given two graphs δ, ϵ on a common set of vertices, to form their pointwise sum $\delta + \epsilon$ defined as $(\delta + \epsilon)_{uv} = \delta_{uv} + \epsilon_{uv}$, their matrix product $\delta\epsilon$ defined as $(\delta\epsilon)_{uv} = \delta_{u-} \cdot \epsilon_{-v}$ (inner product), along with their transitive, symmetric, and reflexive closures, all on the same vertex set.

We would like to consider other operations that combine distinct vertex sets in various ways. The two basic operations we have in mind are the disjoint union and cartesian product of such graphs, along with such variations of these operations as pasting (as not-so-disjoint union), concatenation (as a disjoint union with additional edges from one component to the other), etc.

An efficient way to obtain a usefully large library of such operations is to impose an appropriate categorical structure on the collection of such graphs. In this paper we show how to use enriched categories to provide such structure while at the same time extending the notion of semiring to the more general notion of monoidal category. In so doing we find two layers of categorical structure:

enriched categories in the lower layer, as a generalization of graphs, and ordinary categories in the upper layer having enriched categories for its objects. The graph operations we want to define are expressible as limits and colimits in the upper (ordinary) categories.

We first make a connection between the two universes of graph theory and category theory. We assume at the outset that vertices of graphs correspond to objects of categories, both for ordinary categories and enriched categories. The interesting part is how the edges are treated.

The underlying graph $U(C)$ of a category C consists of the objects and morphisms of C, with no composition law or identities. But there may be more than one morphism between any two vertices, whereas in graph theory one ordinarily allows just one edge. These "multigraphs" of category theory would therefore appear to be a more general notion than the directed graphs of graph theory.

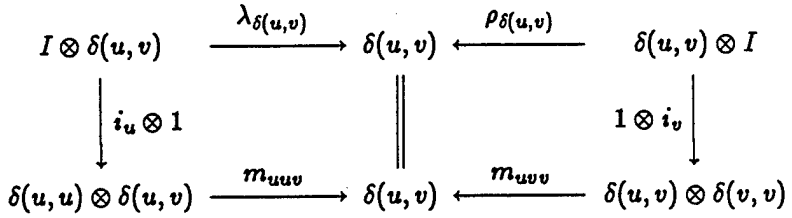
A staple of graph theory however is the label, whether on a vertex or an edge. If we regard a homset as an edge labeled with a set then a multigraph is the case of an edge-labeled graph where the labels are sets. So a multigraph is intermediate in generality between a directed graph and an edge-labeled directed graph.

So starting from graphs whose edges are labeled with sets, we may pass to categories by specifying identities and a composition law, *or* we may pass to edge-labeled graphs by allowing other labels than sets. What is less obvious is that we can elegantly and usefully do both at once, giving rise to enriched categories. The basic ideas behind enriched categories can be traced to Mac Lane [Mac65], with much of the detail worked out by Eilenberg and Kelly [EK66], with the many subsequent developments condensed by Kelly [Kel82]. Lawvere [Law73] provides a highly readable account of the concepts.

We require of the edge labels only that they form a *monoidal category*. Roughly speaking this is a set bearing the structure of both a category and a monoid. Formally a *monoidal category* $\mathcal{D} = \langle D, \otimes, I, \alpha, \lambda, \rho \rangle$ is a category $D = \langle D_0, m, i \rangle$, a functor $\otimes : D^2 \to D$, an object I of D, and three natural isomorphisms $\alpha : c \otimes (d \otimes e) \to (c \otimes d) \otimes e$, $\lambda : I \otimes d \to d$, and $\rho : d \otimes I \to d$. (Here $c \otimes (d \otimes e)$ and $(c \otimes d) \otimes e$ denote the evident functors from D^3 to D, and similarly for $I \otimes d$, $d \otimes I$ and d as functors from D to D, where c, d, e are variables ranging over D.) These correspond to the three basic identities of the equational theory of monoids. To complete the definition of monoidal category we require a certain coherence condition, namely that the other identities of that theory be "generated" in exactly one way from these, see Mac Lane [Mac71] for details.

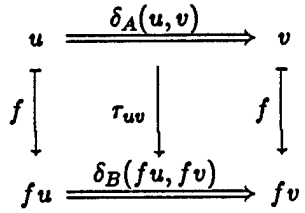
A *$\mathcal{D}$-category*, or (small) *category enriched in* a monoidal category $\mathcal{D}$, is a quadruple $\langle V, \delta, m, i \rangle$ consisting of a set V (which we think of as vertices of a graph), a function $\delta : V^2 \to D_0$ (the edge-labeling function), a family m of morphisms $m_{uvw} : \delta(u,v) \otimes \delta(v,w) \to \delta(u,w)$ of D (the composition law), and a family i of morphisms $i_u : I \to \delta(u,u)$ (the identities), satisfying the following diagrams.

$$\begin{array}{ccccc}
(\delta(u,v) \otimes \delta(v,w)) \otimes \delta(w,x) & & \xrightarrow{\alpha_{\delta(u,v)\delta(v,w)\delta(w,x)}} & & \delta(u,v) \otimes (\delta(v,w) \otimes \delta(w,x)) \\
\Big\downarrow{\scriptstyle m_{uvw} \otimes 1} & & & & \Big\downarrow{\scriptstyle 1 \otimes m_{vwx}} \\
\delta(u,w) \otimes \delta(w,x) & & \xrightarrow{m_{uwx}} \quad \delta(u,x) \quad \xleftarrow{m_{uvx}} & & \delta(u,v) \otimes \delta(v,x)
\end{array}$$



Inspection reveals the first of these as expressing abstractly the associativity of composition and the second as expressing the behavior of identities.

Associated with the notion of $\mathcal{D}$-category is that of $\mathcal{D}$-functor $F : A \to B$ where A and B are $\mathcal{D}$-categories. This is just like an ordinary functor for its object part, mapping objects of A to objects of B via $f : ob(A) \to ob(B)$. The usual morphism part of a functor now becomes a family $\tau_{uv} : \delta_A(u,v) \to \delta_B(fu, fv)$ of morphisms of $\mathcal{D}$:



which compose vertically in the obvious way.

The class of all $\mathcal{D}$-categories and $\mathcal{D}$-functors then forms a (large) category, called $\mathcal{D}$-**Cat**.

The category **Cat** of all small categories can now be seen to be **Set-Cat**. Rendering this abstraction more accessible and appealing is the very pretty case $\mathcal{D} = \mathbf{R}_{\geq 0}^{op} = \langle\langle R_{\geq 0}, \geq\rangle, +, 0\rangle$, reverse-ordered nonnegative reals under addition, for which **R-Cat** becomes the category of (generalized) metric spaces, with the composition law as the triangle inequality and functors as contracting maps [Law73]. Enriched categories first appeared in computer science with $\mathcal{D} = \mathbf{Poset} = \langle Poset, \times, 1\rangle$ [Wan79] yielding order-enriched categories, a natural notion for domain theory. **Poset** itself is definable as (the antisymmetric subcategory of) $\langle\langle\{0,1\}, \to\rangle, \wedge, 1\rangle$-**Cat**, categories enriched in truth-values.

We may now make the connection with semirings. The enriching monoidal category $\langle D, \otimes, I, \alpha, \lambda, \rho\rangle$ has for D_0 the set of edge labels, for $\otimes$ the semiring multiplication, and for its coproduct (which therefore needs to exist in $\mathcal{D}$) the semiring addition. The usual requirement of distributivity of multiplication over addition is met when when $\mathcal{D}$ is *biclosed*—$\otimes$ has a right adjoint in both arguments—with $\mathcal{D}$ *closed* corresponding to one-sided distributivity. (In these situations $\mathcal{D}$ *cartesian* closed is the exception rather than the rule.)

Although the literature has tended to make enriched categories seem if anything more abstract and forbidding than ordinary categories to most computer scientists, this perspective puts enrichment in quite a different light for those familiar with the Floyd-Warshall connection. For $\mathcal{D}$ a preorder with finite coproducts, enriched categories simply become the reflexive and transitive edge-labeled graphs output by the Gauss-Kleene-Warshall-Floyd algorithm. For $\mathcal{D}$ not a preorder, such as **Set** or **Cat**, yielding respectively ordinary categories and 2-categories, the notion becomes more involved

(to which a categoriphobe might say "Ah, so that's the problem") but necessarily so for Gauss's algorithm, whose semiring addition is not idempotent.

This is a nice perspective in its own right, but it becomes considerably more useful when the 2-categorical structure of $\mathcal{D}$-Cat is brought to bear on the description of particular algebras. We illustrate this by applying it to the categorical treatment of Birkhoff's arithmetic of posets [Bir42] and its generalization to other metrics besides the truth-valued metric used for posets. This arithmetic provides a nice abstraction of the sort of concurrency operations we have been advocating [Pra86] to make the "true concurrency" or partially-ordered-time approach more algebraic

Birkhoff defines six operations on posets: addition, multiplication, and exponentiation, each in a cardinal and an ordinal version, as a way of unifying cardinal and ordinal arithmetic. (In the concurrency connection cardinal vs. ordinal corresponds to parallel vs. sequential.) The cardinal operations are conveniently described as universals in **Poset**, the ordinals not quite so conveniently categorically, but 2-categorically ordinal addition becomes just cocomma, indicating that the move from parallel to sequential can usefully be accompanied by a move from categories to 2-categories.

Birkhoff arithmetic admits useful generalizations to other semirings *qua* monoidal categories, suitable for modelling real-valued time in various forms: upper bounds, lower bounds, intervals, and arbitrary sets of reals, each associated with a specific monoidal category, ***but with the definitions of the associated arithmetic operations unchanged.*** These generalizations in turn suggest additional constructs, also definable universally, that would have been meaningless or degenerate in Birkhoff's original framework, but that have useful applications to the specification of real-time processes.

The prospect of a connection with Girard's linear logic obliges us to point out that as both an expansion and a nonconservative extension of the above theory, linear logic with negation is too strong for the purposes of making the connections of this paper, which are more appropriately described as aspects of a fragment of linear logic.

References

[AHU74] A.V. Aho, J. E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass, 1974.

[Bir42] G. Birkhoff. Generalized arithmetic. *Duke Mathematical Journal*, 9(2), June 1942.

[EK66] Samuel Eilenberg and G. Max Kelly. Closed categories. In S. Eilenberg, D. K. Harrison, S. MacLane, and H. Röhrl, editors, *Proceedings of the Conference on Categorical Algebra, La Jolla, 1965*, pages 421–562, Springer-Verlag, 1966.

[Kel82] G.M. Kelly. *Basic Concepts of Enriched Category Theory: London Math. Soc. Lecture Notes. 64*, Cambridge University Press, 1982.

[Law73] W. Lawvere. Metric spaces, generalized logic, and closed categories. In *Rendiconti del Seminario Matematico e Fisico di Milano, XLIII*, Tipografia Fusi, Pavia, 1973.

[Mac65] S. Mac Lane. Categorical algebra. *Bull. Am. Math. Soc*, 71:40–106, 1965.

[Mac71] S. Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[Pra86] V.R. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, February 1986.

[Wan79] M. Wand. Fixed-point constructions in order-enriched categories. *TCS*, 8(1):13–30, 1979.

Finite automata, algorithms and proofs.

(Abstract)

Dauchet, M. & Tison, S.
LIFL (URA 369-CNRS), Université de Lille-Flandres-Artois.
UFR IEEA, 59655 VILLENEUVE D 'ASCQ Cedex FRANCE.
e mail: dauchet@frcitl71.bitnet

Abstract: Generalized automata are a tool for efficient algorithms design...

A short introduction to the usual Tree Automata: The definition is a natural generalization of the usual word automata. (see for example [5,17,26] for the algebraic frame and [22] for the algorithmic point of view). From an "algebraic data types" point of view, states can be seen as sorts of an underlying ordered sorted algebra [2].

Example: Let M the (bottom-up, i.e. frontier-to-root) tree automata defined by the ranked alphabet A, the set S of states, the set F of final states (F is a subset of S) and the set R of rules (each rule can be seen as a signature):

A = {+,*,/,-, suc, 1,0}. +,*,/ and - are binary operators (rank 2); suc is unary (rank 1); 1 and 0 are constants (rank 0). S = {real, int, bool} ;

rules: $0 \rightarrow$ real ; $0 \rightarrow$ int ; $0 \rightarrow$ bool ; $1 \rightarrow$ real ; $1 \rightarrow$ int ; $1 \rightarrow$ bool ;

+(real, real) $\rightarrow$ real ; +(int, int) $\rightarrow$ int ; +(bool, bool) $\rightarrow$ bool ;

*(real, real) $\rightarrow$ real ; *(int, int) $\rightarrow$ int ; *(bool, bool) $\rightarrow$ bool ;

-(real, real) $\rightarrow$ real ; -(int, int) $\rightarrow$ int ; /(real, real) $\rightarrow$ real ; suc(int) $\rightarrow$ int

int $\rightarrow$ real (sorte inclusion is denoted by this special kind of rule without fonction symbol, also called ε-transition)

Intuitively, the automaton M computes from the leaves to the root the sort of a term t. (M[t] denotes the set of states reached by M at the top of t). It fails if a term is bad-sorted and it is non-deterministic because a term can get several sorts. M is said deterministic iff no M[t] contains two states.

Example: M[0] = {real, int, bool}; M[-(1,1)] = {real, int }; M[/(1,+(1,1)] = {real} ;
M[suc(/(1,+(1,1))] is empty (i.e. M fails).

A term t is recognized iff M[t] contains at least a final state. A set of terms is recognizable iff it is the set of terms recognized by some automata.

There exists an algorithm of non-determinism reduction and an algorithm of minimalization; they work like in the word case. So, M' below is the minimal deterministic automaton equivalent to M.

rules of M': $0 \rightarrow$ rib ; $1 \rightarrow$ rib; +(rib,rib) $\rightarrow$ rib ; *(rib,rib) $\rightarrow$ rib ; -(rib,rib) $\rightarrow$ ri; /(rib) $\rightarrow$ r ; succ(rib) $\rightarrow$ i ; +(ri,rib) $\rightarrow$ ri ; +(rib,ri) $\rightarrow$ ri; +(ri,ri) $\rightarrow$ ri; *(ri,rib) $\rightarrow$ ri ; *(rib,ri) $\rightarrow$ ri; *(ri,ri) $\rightarrow$ ri; -(ri,rib) $\rightarrow$ ri ; -(rib,ri) $\rightarrow$ ri; -(ri,ri) $\rightarrow$ ri; -(rib,rib) $\rightarrow$ ri; /(ri) $\rightarrow$ r ; succ(ri) $\rightarrow$ i ; +(i,rib) $\rightarrow$ i ; +(i,ri) $\rightarrow$ i ; +(ri,i) $\rightarrow$ i ; +(r,rib) $\rightarrow$ r etc... etc... . (intuitively, ri can be identified to {real, int } etc... .

Remark that the semantic of M is clear but not that one of M' . It is very usual to translate some algorithm (or to compile some program) to get an efficient but "non-signifiant" algorithm. Here, M' is very efficient in time but the number of rules can exponencialy increase for obvious reasons. The complexity of the non-determinism reduction is coded in other usual problems, as equivalence of two automata. Nevertheless, in usual cases, the number of rules does not increase a lot. Furthermore, it is possible to use dynamic programming à la Morris and Pratt to get linear classes of algorithms (like for recognizion of a term or a subterm). Efficient algorithms are designed using transitive closure, by a way closely related to congruence closures in graphs [27,33].

An important toolbox is available; it links the algebraic point of view (i.e. the specification

point of view) and the algorithmic point of view. Roughly speaking, it generalizes the Kleene theorem: " for any specification, compile the best algorithm". A lot of algebraic tools have been studied. Some are usual from the categorical point of view but sophisticated transducers (a little too much complicated!) have also been introduced to modelize compilation [18]. Most of them have realistic algorithmics properties.

This was a very short sketch of the present situation. Using these tools, and tedious analysis of tree structures (as in formal language theory) we recently solved the following problems (stated in [5], [14], [23], [24], [31]):Decidability of the confluence of ground term rewriting systems [9], [10]; Decidability of the fair termination of ground term rewriting systems [37]; Undecidability of the stability of recognizability under saturated congruence [36]; Undecidability of code problem for non-linear trees and other structures [1]; Decidability of equality of the yields of rational infinite trees [8]; Undecidability of termination of a left-linear rewriting rule [12]. We are designing a software (VALERIAAN, in Prolog [11] ,[[13]) for theorem proving, first and second order reachability problems [31], etc... in some classes of term equations and rewriting. We get an optimised compiler of term rewriting which solves first order reachability in linear time. Roughly speaking, we use dynamic programming, generalizing the famous Morris and Pratt pattern-matching algorithm.

An example of automata used to solve a problem.

The problem of decidability of the confluence of ground term rewriting systems was stated by Huet. We solved it recently by the way sketched below.(see [9], [10])

Definition of the class GTT of ground tree transducers (gtt): let $A = \{(L_i,R_i)|0\leq i\leq n\}$ a finite set of pair of recognizable sets of trees. The corresponding gtt A is defined by

$(u,v)\in A$ iff there exists $t(x1,...,xp)$ $u1,...up$, $v1,...,vp$ such that, $u=t(u1,...,up)$, $v = t(v1,...,vp)$ and for all i $(1\leq i\leq p)$ there exists (Lj_i,Rj_i) in A such that, $ui \in Lj_i$ and $vi \in Rj_i$

1/ Using tree automata technics we prove that : (i) the inverse of a gtt is a gtt; (ii) the composition of two gtt is a gtt; (iii) the precongruence closure of the union of two gtt is a gtt; (iv)the iteration of a gtt is a gtt.

2/ Then it is obvious that the relation R associated to a ground term rewriting system R is a gtt (is suffice to remark that a ground rewriting step is a gtt and use 1/)

3/ R is confluent iff $R \circ R^{-1} \subset R^{-1} \circ R$ (obvious). We then can to code R onto a recognizable tree langage and then reduce the confluence decision to the inclusion of recognizable tree langagues.♦♦

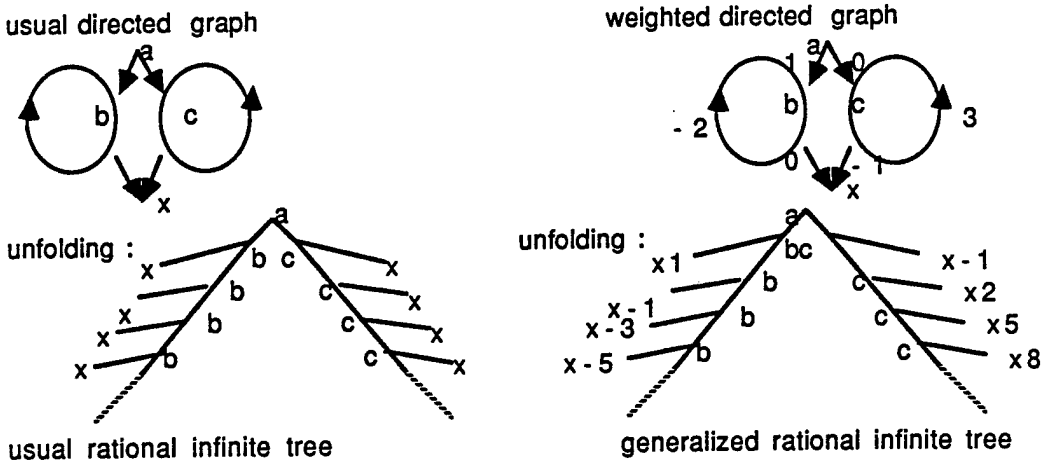
Furthermore, using this characterization of ground term rewriting systems, we get efficient algorithms to solve reachability problems (see VALERIAAN). (Gallier & all. [29,30] recently extended the method of matings due to Andrews to first order languages with equality; they proved that the method of equational matings remains complete when used in conjonction with a restricted kind of E-unification (rigid-unification) using ground rewriting).

Related works and further works: logic and automata.

One of the motivations of tree automata was decision problems of second-order logic. [34] Recent and important works studied connection between logic and automata [6,35].

The general goal is to associate to a logical system a class of automata to get decision properties on the underlying objects (finite or infinite words, trees, graphs)[28,29,30]. This way provides very powerful results, which associate to logical specifications (which can be seen as a veryvery hight level of specification) decision algorithms by the way of automata on different algebraic structures. Unfortunately, the complexity of these algorithms is not realistic. An algebraic and algorithmic study of automata on these structures could provide, at an intermediate level of specification including heuristics, useful tools for an interactive design of efficient algorithms.

Our study of weighted graphs, which generalizes usual infinite rational trees and provide a tool for decision and compexity analysis in Logic Programming, illustrates this way. The algebraic structured can been drawn as following [7,15,16]:



An other way is to extend recognizable sets of trees to recognizable sets of trees with some kind of equality control between subterms [0,2,3]. For example, we considere automata rules which check equalities of subterms [0]. We extend the classes but we keep good decision properties. The results can be used for some decision problems in algebras containing terms with non linear signature.

REFERENCES

[0] Bogaert, B. Tree automata with equality control. Technical report, to appear, LIFL, University of Lille.

[1] Bossut, F., M. Dauchet & B. Warin, "Rationality and recognizability on planar directed acyclic graphs". MFCS'88, Karlsbad, august 88, Lec. Notes Comp. Sci.

[2] Comon, H. "Inductive Proofs by Specification Transformations." Rewriting Technics and Applications, Chapell Hill, North Carolina., april 1989, Lec. Notes Comp. Sci. (Dershowitz ed.), to appear.

[3] Comon, H. Unification et désunification: théorie et application, thèse, Grenoble, 1988

[4] Courcelle, B. Equivalences and transformations of regular systems. Applications to recursive program schemes and grammars, Theor. Comp. Sci. 42 (1986), 1-122.

[5] Courcelle, B. On recognizable sets and tree automata, Theor. Comp. Sci., to appear.

[6] Courcelle, B. Every equational graph is definiable in monadic second order logic, to appear.

[7] Dauchet,M., Devienne Ph.&Lebègue P. (1988). Décidabilité de la terminaison d'une règle de réécriture en tête, *Journées AFCET-GROPLAN*, Bigre+Globule **59**, p. 231-237.

[8] Dauchet, M. & Timmerman, E., Continuous monoids and yields of infinite trees, *RAIRO Inform. Théor.*, **20-3** (1986), 251-274.

[9] Dauchet, M., T. Heuillard, P. Lescanne & S. Tison, Decidability of the Confluence of Ground Term Rewriting Systems, *2nd Symposium on Logic in Computer Science*, New-York, IEEE Computer Society Press (1987), 353-360.

[10] Dauchet, M. & S. Tison, Decidability of confluence in ground term rewriting systems, *Fondations of Computation Theory '85,* Cottbus, Lecture Notes Comp. Sci., **199** (1985), 80-84.

[11]Dauchet, M. & Deruyver, A. "VALERIAAN": Compilation of Ground Term Rewriting Systems and Applications". Rewriting Technics and Applications, Chapell Hill, North Carolina., april 1989, Lec. Notes Comp. Sci. (Dershowitz ed.), to appear.

[12] Dauchet M. Simulation of Turing Machines by a left-linear rewrite rule. Rewriting Technics and Applications, Chapell Hill, North Carolina., april 1989, Lec. Notes Comp. Sci. (Dershowitz ed.), to appear.

[13] Deruyver, A., Gilleron, R., Compilation of term rewriting systems CAAP'89, Lec. Notes Comp. Sci. (Diaz ed.), to appear.

[14] Dershowitz, N. (1987). Termination. *J. Symbolic computation*, **3** p.69-116.

[15] Devienne Ph. & Lebègue P. (1986). Weighted graphs, a tool for logic programming, CAAP 86, Nice, *Springer Lec. notes Comp. Sci.* **214**, p.100-111.

[16] Devienne Ph. (1988). Weighted Graphs, a tool for expressing the Behaviour of Recursive Rules in Logic Programming, in proceeding of *FGCS' 88*, Tokyo. 397-404. Extended paper to appear in TCS.
[17]Eilenberg S. & Wreight,J. "Automata in General Algebras", Information and Control 11, 52-70 (1967).
[18] Engelfriet, J. "Bottom-up and Top-down tree transformations, a comparison,", Math. Systems theory 9, 198-231 (1975).
[19]Fages, F. "Notes sur l'Unification des Termes de Premier Ordre finis ou infinis", Technical Report, INRIA (1986).
[20]Gallier, J.H., Raatz,S. & Snyder, W. "Theorem Proving using Rigid E-Unification: Equational Mating", *2nd Symposium on Logic in Computer Science,* New-York, IEEE Computer Society Press (1987), 338-346.
[21]Gallier,J.H., Narendran, P., J.H., Raatz,S. & Snyder, W. "Theorem Proving Using Equational Matings and Rigid E-Unification.", To appear.
[22]Gecseg.F. & Steinby, M. "Tree Automata", Akademiai Kiado, Budapest, (1984).
[23] Huet, G. & Oppen D. C. (1980). Equations and rewrite rules: A survey, in R. V. Book, ed., New York: Academic Press. *Formal Language Theory: Perspectives and Open Problems*, pp. 349-405.
[24] Jouannaud, J. P. (1987). Editorial of *J. Symbolic computation*, 3, p.2-3.
[25]Martelli, A. & Montanari, U. "An Efficient Unification Algorithm". TOPLAS 4(2), 258-282 (1982).
[26] Mezei J., Wright., Algebraic automata and context-free sets, Information and Control 11 (1967) 3-29
[27]Nelson G & Oppen D.C. "Fast Decision Procedures Based on Congruence Closure". JACM 27(2) , 356-364 (1980).
[28] Nivat M., Perrin D. (eds.), Automata on infinite words, Lec. Notes Comp. Sci., 192, Spronger, 1985.
[29] Nivat M., Perrin D. Ensembles reconnaissables de mots biinfinis, Canad. J. Math. 38, 513-537 (1986).
[30] Nivat M. Infinite words, infinite trees, infinite computations, in "Foundations of Computer Science III.2", (J.W. de Bakker, J. Van Leeuwen, Eds.), Math. Centre Tracts 109, 3-52.
[31] Oyamaguchi M. The reachability problem for quasi-ground ,term rewriting systems, Journal of Information Processing, 9-4, (1986)
[32] Plaisted, Semantic confluence tests and completion methods, Information and Control, 65, 182-215 (1985)
[33]Snyder, E.W., "Efficient Ground Completion: an O(n log n) Algorithm for generating Reduced Sets of Ground Rewrite Rules Equivalent to a Set of Ground Equations." Rewriting Technics and Applications, Chapell Hill, North Carolina., april 1989, Lec. Notes Comp. Sci. (Dershowitz ed.), to appear.
[34] Thatcher, J.W., Wright, J.B. Generalized finite automata with an application to a decision problem of second-order logic, Math. Syst. Theory 2, 57-82 (1968).
[35] Thomas W., Automata on infinite objects., Handbook of Theretical Computer Science, J.V. Leeuwen editor, North-Holland, to appear.
[36] Tison S., J.L. Coquidé, M. Dauchet, about connections between syntactical and computational complexity, to appear.
[37] Tison S. The fair termination is decidable for ground systems, Rewriting Technics and Applications, Chapell Hill, North Carolina., april 1989, Lec. Notes Comp. Sci. (Dershowitz ed.), to appear.

Category-Sorted Algebra-Based Action Semantics

Susan Even and David Schmidt
Computing and Info. Sciences Dept.
Kansas State University
Manhattan, KS 66506 USA
schmidt@cis.ksu.edu

In a series of papers [5,6,7,8,14,15], Mosses and Watt define *action semantics*, a metalanguage for high level, domain-independent formulation of denotational semantics definitions. Action semantics hides details about domain structure (e.g., direct semantics domains vs. continuation semantics domains vs. resumption semantics domains) and coercions (e.g., integers into reals, injections of summands into sum domains) to encourage readability and modifiability. Action semantics notation is of interest as a programming language of itself, for its components (called *actions*) are polymorphic operators that can be composed in three fundamental ways.

We have formulated a model for action semantics based on Reynolds' *category-sorted algebra* [10,12]. In the model, actions are natural transformations, and the composition operators become compositions in a weak "3-category"-like structure. We have used the model to prove the soundness and completeness of a unification-based, decidable, type inference algorithm for action semantics expressions. The proof is notable for its simplicity.

Action Semantics

Actions are combinators; they operate upon *kinds*. (Mosses calls them *facets* [5,7]). A kind is a collection of types; for example, the *functional facet* is the kind of all types that can be used as temporary values in a computation. The types *int*, *bool*, *real*, *bool*×*real*, and so on, belong to the functional facet. (Other facets include the *declarative facet*, which contains types of identifier, value binding, and the *imperative facet*, which contains types of storage structure.) The types in a kind are pre-ordered to reflect subtyping relationships [1,9,11].

Actions are polymorphic mappings on kinds. For example, the action *copy* is the identity mapping on the types in the functional facet, and the action *succ* also maps functional facet values to functional facet values: it increments *int* and *real* values, and it maps non-numbers to a *nonsense* value. Actions exist for all the fundamental operations of programming languages: value passing, arithmetic, binding creation and access, storage allocation and updating, and so on [5,7,8].

Actions can be composed. Arguments to a compound action may pass from one component action to the other sequentially, in parallel, or conditionally. For example, the compound action *copy*; *succ* accepts a functional facet value that is passed sequentially from *copy* to *succ*, and the output is the incremented value. The action *copy* * *succ* accepts a value, which is given in parallel to both *copy* and *succ*. The two results— the value and its successor— are *merged* together into a pair. Finally, *copy* / *succ* accepts a value, which is conditionally given to one of the two actions, based on the typing of the value. The three compositions are used to define derived compositions that describe value flows found in programming languages. For example, * and ; are combined to describe the flows of bindings and storage, respectively, in command sequencing.

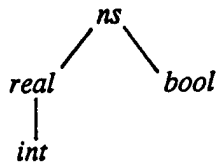
Coercions of arguments and results of actions occur implicitly and naturally (that is, the placement of coercions does not affect the output of an action). For example, if an *int* argument is given to *succ*, but context demands a *real* answer, an implicit coercion can occur either on the argument or on the answer and the

result is the same in either case.

An action semantics may possess many kinds, and the kinds can themselves be preordered. The composition operators respect the subkinding.

Category-Sorted Algebra

Action semantics demands a model that supports the Scott-domain thoery upon which denotational semantics is based. Scott-domains, subdomain relationships, and polymorphic operations are naturally described within category-sorted algebra (*csa*) [10,12]. The appendix gives a precise definition of a csa; here we supply an example of one in the form of a sample functional facet. Let {*copy*, *succ*} be a set of action names and let A be the poset of type names:



Let T_{copy} be the identity operation on the poset, and let T_{succ} map *int* to *int*, *real* to *real*, and *bool* and *ns* to *ns*. (T_{copy} and T_{succ} are the "typing functions" for actions *copy* and *succ*.) Now, $(A, \{T_{copy}, T_{succ}\})$ forms a single-sorted algebra (*ssa*); the ssa plus the operator set {*copy*, *succ*} form a *signature* for a csa.

The *carrier* of the csa is a functor $F: A \Rightarrow Pdom$ (*Pdom* is the category of predomains, i.e., "bottomless cpos") that maps *int* to **Z**, *real* to **R**, *bool* to **B**, and *ns* to **1** (the terminal object in *Pdom*). The functor interprets the type names and the coercion mappings between them. The operators are interpreted as natural transformations: the *copy* operator becomes the identity natural transformation in $F \dot{\rightarrow} F \circ T_{copy}$, and the *succ* operator becomes a natural transformation in $F \dot{\rightarrow} F \circ T_{succ}$. The natural transformations respect the coercion maps established by the carrier.

Other facets are defined similarly. Indeed, the complete structure of action semantics is defined as a csa of a csa, where the first csa defines the facet hierarchy (of which the poset seen above is part), and the second, many-sorted, csa defines the interpretation of the facets (of which the csa seen above is part).

The csa framework accommodates direct and continuation-style denotational semantics for actions. An action like *succ* can be defined as a natural family of direct semantics functions in *Expressible-Value* $\rightarrow$ *Expressible-Value* or as a natural family of continuation semantics functions in (*Expressible-Value* $\rightarrow$ *Answer*) $\rightarrow$ (*Expressible-Value* $\rightarrow$ *Answer*).

Applications

Action semantics expressions are uncluttered by typing annotations; nonetheless, such annotations are invaluable to analysis and implementation. We have defined a unification-based type inference algorithm that annotates an action expression with a typing scheme that indicates its sensical behavior in its context of use.

The algorithm assigns primitive type schemes to primitive actions. For example, the actions *copy* and *succ* are given type schemes:

copy: $\theta \rightarrow \theta$
succ: $\theta \rightarrow \theta$ if $\theta \leq real$

The second scheme says that *succ* has an answer type that matches its argument type if the argument type θ satisfies the constraint $\theta \leq real$ [3,4].

A composed action expression has its type scheme inferred from the types of its components. For actions:

$a_1 : \sigma_1 \to \tau_1$ *if* C_1

and

$a_2 : \sigma_2 \to \tau_2$ *if* C_2

the algorithm infers:

$(a_1; a_2) : U\sigma_1 \to U\tau_2$ *if* $U(C_1 \cup C_2)$

where U is the most general unifier of τ_1 and σ_2. Other forms of composition are treated similarly.

Let action a have a typing function T_a in the csa interpretation. A typing scheme $a : \sigma \to \tau$ *if* C is *sound* if, for all substitutions U such that $U\sigma$ is a completely instantiated type name and $U(C)$ is a set of completely instantiated constraints that hold true, $T_a(U\sigma) = U\tau$. The scheme is *complete* if, for all types t, $T_a(t) \neq ns$ implies there exists a substitution U such that $U\sigma = t$ and $U(C)$ is a completely instantiated set of constraints that hold true.

We have proved the soundness and completeness of the type inference algorithm. No complex proof-theoretic techniques are needed to establish the results, because the csa model provides simple, significant information in the form of the T_a typing functions. Further, the model discourages formulation of a type inference algorithm that attempts to insert explicit coercions. Since actions are natural transformations, coercions are unnecessary; the actions *must* respect the subtyping ordering, whether coercions are used or not. Finally, the inference algorithm is decidable, since natural transformations are "shallow universally quantified" (like the polymorphic operators in ML). Thus, many of the sticky problems found in type inference for programming languages with polymorphism and subtyping are avoided by selection of the csa model.

We have also implemented a prototype interpreter for action semantics along the lines of [14] but with a more careful treatment of the facet flows to actions [2].

References

[1] Cardelli, L., and Wegner, P. On understanding types, data abstraction, and polymorphism. *Computing Surveys* 17-4 (1985) 471-522.

[2] Even, S. An implementation of action semantics. M.S. report, Computer Science Dept., Iowa State Univ., Ames, Iowa, 1987.

[3] Jategaonkar, L., and Mitchell, J. ML with extended pattern matching and subtypes. Proc. 1988 ACM Conf. on LISP and Functional Programming, Snowbird, Utah, July 1988, pp. 198-211.

[4] Mitchell, J. Coercion and type inference. Proc. 11th ACM Symp. on Prin. of Prog. Lang., Salt Lake City, Utah, 1984, pp. 175-186.

[5] Mosses, P. Abstract semantic algebras! In *Formal Description of Programming Concepts II*, D. Bjoerner, ed., North-Holland, Amsterdam, 1983, pp. 45-72.

[6] ___. A basic abstract semantic algebra. In *LNCS 173: Semantics of data types*, Springer, Berlin, 1984, pp. 87-108.

[7] ___. The modularity of action semantics. To appear in *SDF Benchmark Series in Computational Linguistics- Workshop II*, MIT Press, Cambridge.

[8] ___. Unified algebras and action semantics. To appear in Proc. STACS89, Paderborn, Feb. 1989, Springer LNCS.

[9] Oles, F. Type algebras, functor categories, and block structure. In *Algebraic Methods in Semantics*, M. Nivat and J. Reynolds, eds. Cambridge Univ. Press, Cambridge, 1985.

[10] Reynolds, J. Using category theory to design implicit conversions and generic operators. In *LNCS 94: Semantics-Directed Compiler Generation*, N. Jones, ed. Springer, 1980, pp. 211-258.

[11] ___. The essence of Algol. In *Algorithmic Languages*, J. deBakker and J.C. vanVliet, eds., Noth-Holland, Amsterdam, 1981, pp. 345-372.

[12] ___. Semantics as a design tool. Course lecture notes, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, PA, 1988.

[13] Stansifer, R. Type inference with subtypes. Proc. 15th ACM Symp. on Prin. of Prog. Lang., San Diego, CA, 1988, pp. 88-97.

[14] Watt, D. Executable semantic descriptions. *Software: Practice and Experience*, 16 (1986) 13-43.

[15] ___. An action semantics of standard ML. In *LNCS 298: Mathematical Foundations of Programming Semantics*, M. Main, et. al., eds., Springer, Berlin, 1987, pp. 572-598.

Appendix

The notation and definitions are from [12]. Let $S=(Ob(S), Mor(S), \circ)$ be a category; assume that S has finite products.

Definition: An Ω-*signature* is a pair (Ω, ar), where Ω is a set of operators, and $ar: \Omega \to N$ is a function that gives the arity of the operators.

Definition: A (single-sorted) Ω-*algebra (based on S)* is a pair $A = (|A|, \{ A_\omega \mid \omega \in \Omega \})$, where $|A| \in Ob(S)$ is the carrier of the algebra, and for each $\omega \in \Omega$, operation $A_\omega: |A|^{ar\,\omega} \to |A|$ is in $Mor(S)$.

Definition: An Ω-T *signature* is a triple (Ω, ar, T), where Ω and ar are defined as above, and T is an Ω-algebra based on *PreO*. (*PreO* is the category of preordered sets and monotone mappings.)

Definition: An Ω-T *category-sorted algebra (based on S)* is a pair $A = (|A|, \{ A_\omega \mid \omega \in \Omega \})$, where $|A|: |T| \Rightarrow S$, the carrier of A, is a functor from $|T|$, treated as a category in the usual way, to category S; and for each $\omega \in \Omega$, operation $A_\omega: |A|^{ar\,\omega} \dot{\to} |A| \circ T_\omega$ is a natural transformation, where T_ω is treated as an endofunctor on $|T|$.

The above definitions easily generalize to many-sorted algebras and category many-sorted algebras.

THE DE BRUIJN ALGEBRA

Didier VIDAL

C.R.I.N. and L.I.F.L.*

e-mail : dvidal@frcitl71.bitnet

ABSTRACT : The substitution process of de Bruijn calculus is analysed with an equationnal theory, called the "de Bruijn Algebra". Two optimisations are described by equations, which can be oriented into term rewriting systems : parallel substitution and a "labelled" substitution delaying the recoding of the argument free variables.

KEYWORDS : λ-calculus, de Bruijn calculus, substitution, algebra, equational theory, term rewriting system.

Introduction

Implementations of functional languages rely on λ-calculus as a firm mathematical ground. Lambek [3] showed a close relation between typed λ-calculus and Cartesian Closed Categories (CCC in the sequel), and later, Curien [2] used Lambek's formalism to show that λ-terms in de Bruijn's notation [1] could be translated into CCC-terms. This approach led to an efficient implementation of the language ML, (originally developped at the University of Edinburgh). The Categorical Abstract Machine — on which this implementation is based — performs weak reductions and takes advantage of the pairing of functions and of the polymorphism of the "categorical combinators".

In [4], we have studied an implementation of a functional language based on *strong reduction* : programs are untyped λ-terms internally coded with de Bruijn's notation, and its semantics is given by the head normal form.

In λ-calculus theory, substitution is treated as a *one-step* process, and this is unsuitable for practical implementations. This remark led us to study substitution more carefully. We have derived various substitution algorithms from our algebraïc approach which formalize this process and improve it.

We shall introduce an abstract algebra, that we have called the "de Bruijn Algebra" — dBA for short — which is directly inspired by CCC and Curien's work. This algebra defines an *equational theory* where the substitution process is entirely *decomposed* and *simulated* by its axioms. Moreover, we shall not be restricted by typed terms : we can forget about typed theory, and formal computations in dBA will serve the untyped λ-calculus theory as well.

* C.R.I.N. UA CNRS 262, BP 239, F-54506 VANDŒUVRE CEDEX, and L.I.F.L. UA CNRS 369, UFR d'I.E.E.A. Bât.M3. F-59655 VILLENEUVE D'ASQ CEDEX

Definition

We give the "strong rules system" of [2] (p.10) with our notations (and name). Then, the standard substitution algorithm in de Bruijn's notation is recalled.

The *de Bruijn Algebra* is defined as follows:

0-arity operators (i.e. constants): π, π', id,

unary operators: $\lambda.$, $\#$ and $+$,

binary operators: $\cdot$ (this dot will in fact be ommitted) and $\circ$.

If $f \in dBA$, the results of each of the three unary operators are respectively noted: $(\lambda.f)$, $f^{\#}$ and f^{+}. With these notations, the axioms of dBA are:

(1) $(\lambda.f)g = f \circ g^{\#}$
(2) $(f \circ g) \circ h = f \circ (g \circ h)$
(3) $(\lambda.f) \circ g = \lambda.(f \circ g^{+})$
(4) $(fg) \circ h = (f \circ h)(g \circ h)$
(5) $\pi \circ f^{+} = f \circ \pi$
(6) $\pi' \circ f^{+} = \pi'$
(7) $\pi \circ f^{\#} = id$
(8) $\pi' \circ f^{\#} = f$
(9) $id \circ f = f$
(10) $f \circ id = f$

Some intuitive feeling can be caught from CCC : each of these axioms are indeed simple theorems of CCC theory. $\circ$ is borrowed from the composition of arrows, id comes from the identity arrow, π and π' correspond respectively to the first and second projections of cartesian categories. If $f: A_1 \to A_2$, then $f^{+}: A_1 \times B \to A_2 \times B$ is $\langle f \circ \pi, \pi' \rangle$, and $f^{\#}: A_1 \to A_1 \times A_2$ is $\langle id_{A_1}, f \rangle$. In a cartesian *closed* category, we have two maps $App: (B^A) \times A \to B$ and $\lambda.: (A \times B \to C) \to (A \to C^B)$, such that, for any $f: A \times B \to C$, $\lambda.f$ is the unique map satisfying $App \circ (\lambda.f)^{+} = f$. Now, if fg is defined as $App \circ \langle f, g \rangle$, (1), (3) and (4) above are easy consequences of these definitions. For example, let's first notice that $f^{+} \circ g^{\#} = \langle f, g \rangle$ and $f^{+} \circ g^{+} = (f \circ g)^{+}$ in any CCC, then $f \circ g^{\#} = App \circ (\lambda.f)^{+} \circ g^{\#} = App \circ \langle \lambda.f, g \rangle = (\lambda.f)g$, and $f \circ g^{+} = App \circ (\lambda.f)^{+} \circ g^{+} = App \circ ((\lambda.f) \circ g)^{+} \Rightarrow (\lambda.f) \circ g = \lambda.(f \circ g^{+})$, by unicity.

REMARK: In dBA we could not, for example, deduce (4) from the other axioms, contrasting with the proof in a cartesian closed category using uniqueness of $\langle f, g \rangle = h$ such that $\pi \circ h = f$ and $\pi' \circ h = g$. The same is true for (3), which must be taken here as an axiom. Our equational theory is weaker than CCC theory (which is of course not equational) but strong enough for λ-calculus purpose, as we shall see shortly. Moreover, we have eliminated all couples explicitly since they are not present either in λ-calculus.

When the integer-coded bound variable $\underline{n}$ is interpreted by $\pi' \circ \pi \circ \cdots \circ \pi$ (with n copies of π), as in [2], a λ-term in de Bruijn's notation corresponds, *without changing its syntax*, to an element in dBA. The image of λ-terms in dBA by this injective morphism will be noted dB. For the next two lemmas, we shall introduce some more notations:

NOTATIONS: (*i*) $(f^{\#})^{+} = f^{\#+}$, $(f^{+})^{+} = f^{++}$ etc. and $f^{\#+\cdots+} = f^{[k]}$, if there are k copies of $+$ (in particular $f^{[0]} = f^{\#}$).
(*ii*) $(\pi^d)^{+\cdots+} = \pi^{d,k}$ (k copies of $+$).

LEMMA. *With the preceding notations, for all $k \geq 0$ and $n \geq 0$, we have :*

$$\underline{n} \circ f^{[k]} = \begin{cases} \underline{n} & \text{if } n < k, \\ f \circ \pi^n & \text{if } n = k, \\ \underline{n-1} & \text{if } n > k. \end{cases}$$

LEMMA. *For any $f \in dB$ and any integer $d \geq 0$, the term $f \circ \pi^d$ can be reduced to a term $g \in dB$ by the following rules :*

$$(\lambda.f) \circ \pi^{d,k} = \lambda.(f \circ \pi^{d,k+1}),$$
$$(f_1 f_2) \circ \pi^{d,k} = (f_1 \circ \pi^{d,k})(f_2 \circ \pi^{d,k}),$$
$$\underline{n} \circ \pi^{d,k} = \begin{cases} \underline{n} & \text{if } n < k, \\ \underline{n+d} & \text{if } n \geq k. \end{cases}$$

It is easily checked that, for any $f \in dB$, the new term produced by the reduction of $f \circ \pi^n$ is identical to f except for the *free variables* which are all recoded by increasing their code by n.

By definition, substitution *of a dB-term g in an other dB-term f consists in reducing $f \circ g^{\#}$.*

We have the following main result :

THEOREM. If M and N are two λ-terms, and $\bar{M}$ and $\bar{N}$ their respective counterparts in dBA, then :

$$dBA \vdash \bar{M} = \bar{N} \iff \lambda \vdash M = N$$

Substitution improved

We shall show how to prove the correctness of two optimisations of the standard substitution algorithm with the help of dBA. In fact, from equalities in dBA, we get the recursive algorithms we are looking for.

1 — Parallel substitution

We want to get here an algorithm for parallel substitution.

NOTATIONS: for $k \geq 0$, let $\vec{g}^{[k]} = g_1^{[\ell+k-1]} \circ \cdots \circ g_\ell^{[k]}$.

LEMMA. *(i)* $(\lambda^\ell.f) g_1 g_2 \cdots g_\ell = f \circ \vec{g}^{[0]}$.

(ii) $(\lambda.f) \circ \vec{g}^{[k]} = \lambda.(f \circ \vec{g}^{[k+1]})$,

(iii) $(f_1 f_2) \circ \vec{g}^{[k]} = (f_1 \circ \vec{g}^{[k]})(f_2 \circ \vec{g}^{[k]})$,

(iv) $$\underline{n} \circ \vec{g}^{[k]} = \begin{cases} \underline{n} & \text{if } n < k, \\ g_{\ell-n+k} \circ \pi^k & \text{if } k \leq n < k+\ell, \\ \underline{n-\ell} & \text{if } n \geq k+\ell. \end{cases}$$

(i) shows what we want to compute and is easily checked. (ii), (iii) and (iv) give a deterministic algorithm, when interpreted by rewrite rules (orienting equalities from left to right) on dB : only one of the three rules can be applied to a given dB-term, depending on its structure (i.e abstraction, application, or variable).

2 — *Labelled substitution*

Our aim is to delay the recoding of the free variables of a substituted argument (this recoding is necessary in the standard algorithm), so that sharing can be acheived.

We are going to define substitution on a larger subset of dBA than dB. Indeed, "dB-terms not yet recoded" are of the form $f \circ \pi^d$, with $f \in dB$ and $d \geq 0$. These terms will be called "labelled terms" and the subset of dBA they form will be noted ldB. In dB, redexes are simply $(\lambda.f)g$, now we have to consider also those of the form $\big((\lambda.f) \circ \pi^d\big)g$. The new substitution will perform at the same time the recoding of $(\lambda.f)$ and the substitution of the argument g. Moreover, recoding of g will not be computed, but delayed.

NOTATIONS: (*i*) $(\pi^d)^+ \circ g^{\#} = g^{(d)}$,

(*ii*) $g^{(d)k} = g^{(d)+\cdots+}$ if there are k copies of $+$.

LEMMA. *(i)* $\big((\lambda.f) \circ \pi^d\big)g = f \circ g^{(d)}$,

(ii) $(f_1 f_2) \circ g^{(d)k} = \big(f_1 \circ g^{(d)k}\big)\big(f_2 \circ g^{(d)k}\big)$,

(iii) $(\lambda.f) \circ g^{(d)k} = \lambda.\big(f \circ g^{(d)(k+1)}\big)$,

(iv) $\underline{n} \circ g^{(d)k} = \begin{cases} \underline{n} & \text{if } n < k, \\ g \circ \pi^n & \text{if } n = k, \\ \underline{n+d-1} & \text{if } n > k, \end{cases}$

(v) $\pi^{d'} \circ g^{(d)k} = \begin{cases} g^{(d)(k-d')} \circ \pi^{d'} & \text{if } d' \leq k, \\ \pi^{d'+d-1} & \text{if } d' > k. \end{cases}$

Again, from these equalities converted into a term rewriting system on ldB, we get a substitution algorithm on labelled terms. The last case (v) shows how to deal with substitution into a labelled term, say $f \circ \pi^{d'}$: if the label d' is greater than the depth k where g is to be substituted, it is clear that there will be no free variable with a code equal to k and consequently no occurrence where to substitute g, hence, in that case, we get a simple result viz. $\pi^{d+d'-1}$ (as it can be checked).

REMARK : The two algorithms could be mixed to produce a "parallel and labelled" substitution (see [4]).

Conclusion

We have presented an abstract algebra, the *de Bruijn Algebra*, which contains the set of λ-terms in de Bruijn's notation, and also other interesting terms like the so-called "labelled (de Bruijn) terms". The substitution process can be investigated in great details with this algrebraïc approach. We have indicated how various substitution algorithms can be deduced from the standard one and how to improve it. Other results converging to an efficient implementation of λ-calculus can be found in [4]. They are based on the notion of *relocalisation* of redexes, which allows to interpret the integer codes of the variables as offset-addresses in a stack of arguments and prove the correctness of abstract machines.

Finally, let's mention that if one is interested by η-reduction, the following axiom would have to be considered : $\lambda.(f \circ \pi)\pi' = f$.

References

[1] **de Bruijn, N.G.**, *Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem*, Indag. Math. 34, pp 381–392, 1972.

[2] **Curien P-L.**, *Categorical Combinators, Sequential Algorithms and Functional Programming*, Research Notes in Theoretical Computer Science, Pitman, London, 1986.

[3] **Lambek J.**, *From λ-calculus to Cartesian Closed Categories*, in "To H.B. Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism", ed J.P. Seldin and J.R Hindley, Academic Press (1980).

[4] **Vidal D.**, *Nouvelles notions de réduction en λ-calcul*, Thèse de Doctorat de l'Université de Nancy, 1989.

A Game Characterization of the Observational Equivalence of Processes (Extended Abstract)

M. Halit Oguztuzun
Department of Computer Science
University of Iowa
Iowa City, IA 52242

This preliminary work is concerned with the characterization of the observational equivalence of processes in model-theoretic terms. First, the Ehrenfeucht game is extended by introducing a condition of "compatibility", and then it is shown that the equivalence induced by the extended game with an appropriate notion of compatibility coincides with observational equivalence. Second, a subclass of first-order languages is defined by "translating" this specific compatibility notion into a syntactical constraint. It is conjectured that the language thus obtained corresponds to the extended game in the sense that a first-order language corresponds to the original game.

The Ehrenfeucht game is played by two players, Player I and Player II, given two similar (relational) structures, $\mathcal{A}$=<A,$\{R_i|i \in I\}$ > and $\mathcal{B}$=<B,$\{S_i|i \in I\}$ > where I is some index set. With $\mathcal{A}$ [resp.$\mathcal{B}$] we associate a reflexive relation, $C_{\mathcal{A}}$ [resp.$C_{\mathcal{B}}$] on A [resp.B]. We call them the ***compatibility relations.*** Let n be a fixed natural number. We then denote the extended Ehrenfeucht game by $G_n(\mathcal{A},C_{\mathcal{A}},\mathcal{B},C_{\mathcal{B}})$. A play of this game consists of n rounds, each of which is played as follows. First, Player I chooses an element of either A or B. In response, Player II picks an element of the other structure. Each element must be "compatible" with the element chosen from that set at the previous round. More precisely, let c_i be the element of A [or B] chosen at round i. In the next round, for any player to choose an element c_{i+1} of A [resp.B], $c_i C_{\mathcal{A}} c_{i+1}$ [resp.$c_i C_{\mathcal{B}} c_{i+1}$] must hold. At the end of the play, we have $< a_1,\ldots,a_n >$, the sequence of elements chosen from A, and $< b_1,\ldots,b_n >$, the sequence of elements chosen from B. Player II ***wins the play of the game*** iff the correspondence between a_i and b_i $(i = 1,\ldots,n)$ is an isomorphism with respect to the relations of $\mathcal{A}$ and $\mathcal{B}$. Otherwise Player I wins.

We define a relation on similar structures by Player II having a winning strategy. This turns out to be an equivalence relation. More precisely, let $\mathcal{A}$ and $\mathcal{B}$ be two similar structures. We say that $\mathcal{A}$ is G_n-***equivalent to*** $\mathcal{B}$ (w.r.t. $C_{\mathcal{A}}$ and $C_{\mathcal{B}}$) iff Player II has a winning strategy in the game $G_n(\mathcal{A},C_{\mathcal{A}},\mathcal{B},C_{\mathcal{B}})$. We say that $\mathcal{A}$ is G-***equivalent to*** $\mathcal{B}$ iff $\mathcal{A}$ is G_n-equivalent to $\mathcal{B}$ for all n (given $C_{\mathcal{A}}$ and $C_{\mathcal{B}}$).

We model a process as a ***synchronization tree (st).*** An *st* is a rooted, unordered, labelled, finitely-branching tree [4]. We can view an *st* as the unfolding of a nondeterministic state transition system with "silent" moves. Formally, we represent an *st* A as a structure A=<A,$\{R_\mu|\mu \in \Lambda;\tau\}, a_0$ >,where A is a countable set (of nodes, or states), Λ is a finite set (of labels) and $\tau \notin \Lambda$, R_μ $(\mu \in \Lambda;\tau)$ are binary relations on A (arcs, or transitions), and $a_0 \in$A (the root, or initial state). (*Notation:* $\Lambda;\tau = \Lambda \cup \{\tau\}$.) The silent transition R_τ has to be reflexive. So we have a self-loop labelled τ at each node.

We define the observational equivalence on *sts* (denoted $\approx$) as follows.
Let two *sts* A and B be given as above. Then, let $\Rightarrow^\mu$ be the closure of R_μ under left and right relational compositions with R_τ, for $\mu \in \Lambda;\tau$. We identify an *st* with its root. Now the definition:

$A \approx_0 B$. $A \approx_{k+1} B$ iff
(i) $A \Rightarrow^{\mu} A'$ implies $B \Rightarrow^{\mu} B'$ and $A' \approx_k B'$ for some B'; and
(ii) $B \Rightarrow^{\mu} B'$ implies $A \Rightarrow^{\mu} A'$ and $A' \approx_k B'$ for some A'.
$A \approx B$ iff for all k $A \approx_k B$.

Given two *sts* A and B, we consider the game $G_n(\mathcal{A}, \leadsto_{\mathcal{A}}, \mathcal{B}, \leadsto_{\mathcal{B}})$ where $\mathcal{A} = <A, \{\Rightarrow^{\mu} | \mu \in \Lambda; \tau\} >$ where $\Rightarrow^{\mu}$ is defined as above ($\mu \in \Lambda; \tau$), and the compatibility relation $\leadsto_{\mathcal{A}}$ is defined as $\cup_{\mu \in \Lambda; \tau} \Rightarrow^{\mu}$. Similarly, we define $\mathcal{B}$ and $\leadsto_{\mathcal{B}}$ for B.

Theorem: Given two synchronization trees A and B. $A \approx B$ iff the structures $\mathcal{A}$ and $\mathcal{B}$ defined above are G-equivalent (w.r.t. $\leadsto_{\mathcal{A}}$ and $\leadsto_{\mathcal{B}}$).

Now, consider a first-order language $\mathcal{L}$ with a finite number of two-place predicate symbols and a constant symbol, without equality. The predicate symbols are to be interpreted as relations $\Rightarrow^{\mu}$ ($\mu \in \Lambda; \tau$), and the constant symbol is to be interpreted as the root. Let φ be a formula of $\mathcal{L}$ which is not tautologically false. Consider a formula φ' of $\mathcal{L}$ which is logically equivalent to φ and in the prenex-disjunctive normal form. Let ψ be a disjunct of the matrix of φ'. Define a relation $>_{\psi}$ on the set of variables and the constant symbol as follows: $x >_{\psi} y$ iff ψ has an atomic formula Pxy or the negation of it as a conjunct. If $>_{\psi}$ is merge-free for every disjunct ψ of the matrix then we call φ *special*. If φ is tautologically false then we take it as special. (We call a binary relation R *merge-free* iff xRz and yRz implies $x = y$.) The subset $\mathcal{L}_*$ of $\mathcal{L}$ is defined so that the formulas of $\mathcal{L}_*$ are exactly the special formulas of $\mathcal{L}$. We say that two structures $\mathcal{A}$ and $\mathcal{B}$ are *elementarily equivalent w.r.t.* $\mathcal{L}_*$ iff for any closed formula σ of $\mathcal{L}_*$, $\mathcal{A}$ satisfies σ iff $\mathcal{B}$ satisfies σ.

Conjecture: Let $\mathcal{A}$ and $\mathcal{B}$ be two similar structures having a finite number of binary relations. Let their respective compatibility relations, $\leadsto_{\mathcal{A}}$ and $\leadsto_{\mathcal{B}}$ be defined as above. $\mathcal{A}$ and $\mathcal{B}$ are G-equivalent iff they are elementarily equivalent w.r.t. $\mathcal{L}_*$.

Related Work: The idea of observational equivalence is prevalent in Milner's work on the Calculus of Communicating Systems, see, e.g. [4,5]. The definition we adopt here is called the "weak observational equivalence" in [1]. This reference is a comparative study of several operational and logical notions of process equivalence. Hennessy and Milner [3] proposed a modal language to characterize observational equivalence. The game characterization of the elementary equivalence of similar finitary structures is due to Ehrenfeucht [2].

Acknowledgements

The help received from George Nelson was indispensible. Discussions with M.N. Muralidharan led to improvements on several points. Thanks are also due to Teo Rus for his encouragement.

References

[1] Brookes,S.D. and Rounds,W.C. Behavioral equivalence relations induced by programming logics. In *Proc.ICALP 83*,LNCS 154,Springer-Verlag,Berlin,1983, 97-108.

[2] Ehrenfeucht,A. An application of games to the completeness problem for formalized theories. *Fundamenta Mathematicae*,XLIX (1961),129-141.

[3] Hennessy,M. and Milner,R. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM* 32,1 (January 1985),137-161.

[4] Milner,R. *A Calculus of Communicating Systems.* LNCS 92, Springer-Verlag,Berlin,1980.

[5] Milner,R. Lectures on a calculus for communicating systems. In *Proc. Seminar on Concurrency,July 1984*,LNCS 197,Springer-Verlag, Berlin,1985,197-220.

Refinement in branching time semantics

Rob J. van Glabbeek and W.Peter Weijland
Centre for Mathematics and Computer Science
P.O.Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract: In this paper we consider branching time semantics for finite sequential processes with silent moves. We show that Milner's notion of *observation equivalence* is not preserved under refinement of actions, even when no interleaving operators are considered; however, the authors' notion of *branching bisimulation* is.

Note: This paper is sponsored in part by Esprit profect no.432, METEOR.

INTRODUCTION

Virtually all semantic equivalences employed in theories of concurrency are defined in terms of *actions* that concurrent systems may perform (cf. [1-7]). Mostly, these actions are taken to be *atomic*, meaning that they are considered not to be divisible into smaller parts. In this case, the defined equivalences are said to be based on *action atomicity*.

However, in the top-down design of distributed systems it might be fruitful to model processes at different levels of abstraction. The actions on an abstract level then turn out to represent complex processes on a more concrete level. This methodology does not seem compatible with non-divisibility of actions and for this reason, PRATT [7], LAMPORT [4] and others plead for the use of semantic equivalences that are not based on action atomicity.

As indicated in CASTELLANO, DE MICHELIS & POMELLO [2], the concept of action atomicity can be formalised by means of the notion of *refinement of actions*. A semantic equivalence is *preserved under action refinement* if two equivalent processes remain equivalent after replacing all occurrences of an atomic action a by a more complicated process r(a). In particular, r(a) may be a sequence of two actions a_1 and a_2. An equivalence is strictly based on action atomicity if it is not preserved under action refinement.

In a previous paper [3] the authors argued that MILNER's notion of observation equivalence [5] does not respect the branching structure of processes, and proposed the finer notion of *branching bisimulation equivalence* which does. In this paper we moreover find, that observation equivalence is not preserved under action refinement, whereas branching bisimulation equivalence is.

1. PROCESS GRAPHS

As a simple model, let us represent a process by a *state transition diagram* or *process graph*. Such a graph has a node for every one of the possible states of the process, and has arrows between nodes to indicate whether or not a state is accessible from another. Furthermore, these arrows (directed edges) are labelled, with labels from $A \cup \{\tau\}$, where $A = \{a,b,c,...\}$ is some set of *observable signals*, and τ stands for a *silent step* (cf. [5]).

DEFINITION 1.1 A *process graph* is a connected, rooted, edge-labelled and directed graph.

In an edge-labelled graph, one can have more than one edge between two nodes as long as they carry different labels. A rooted graph has one special node which is indicated as the root node. Graphs need not be finite, but in a connected graph one must be able to reach every node from the root node by following a finite path. If r and s are nodes in a graph, then $r \rightarrow^a s$ denotes an edge from r to s with label a (it is also used as a proposition stating that such an edge exists). In this paper we limit ourselves to processes represented by finite, non-trivial process graphs. A graph is *finite* if it is acyclic and contains only finitely many nodes and edges; it is *trivial* if it contains no edges at all. The set of non-trivial, finite process graphs will be denoted by G.

In order to turn G into an algebraic structure, it is possible to define binary operators '+' and '·' for alternative and sequential composition. For any two graphs g and h the process graph (g + h) is obtained by simply identifying their root nodes, whereas (g·h) - often written as just (gh) - can be found by identifying the root node of h with all endnodes of g. Furthermore, constants from $A \cup \{\tau\}$ are interpreted as one-edge graphs, carrying the constant as their edge-label. The algebraic structure allows us to study equational theories that emerge from any defined equivalence on G. For instance, in branching time semantics, one often considers *observation congruence* (cf. MILNER [5]) - written as $\approx^c$ - as a deciding criterion for equality in observable behaviour. Let us write $r \Rightarrow r'$ for a path from r to r' consisting of an arbitrary number (≥ 0) of τ-edges. Then its definition can be rephrased as:

DEFINITION 1.2 Two graphs g and h are *observation equivalent* if there exists a symmetric relation $R \subseteq \text{nodes}(g) \times \text{nodes}(h) \cup \text{nodes}(h) \times \text{nodes}(g)$ (called a τ-*bisimulation*) such that:

1. The roots are related by R.
2. If R(r,s) and $r \rightarrow^a r'$ ($a \in A \cup \{\tau\}$), then either $a=\tau$ and R(r',s), or there exists a path $s \Rightarrow s_1 \rightarrow^a s_2 \Rightarrow s'$ such that R(r',s').

Furthermore, g and h are *observation congruent* if we also have that

3. (root condition) Root nodes are related with root nodes only.

The root condition was first formulated by BERGSTRA & KLOP [1], and serves to turn the notion of observation equivalence into a congruence with respect to the operators + and ·. It can be proved that observation equivalence and observation congruence are equivalence relations on G, and that the latter is the coarsest congruence contained in the former (cf. [5,1,3]). It was shown in [1] that with respect to closed terms the model $G/\approx^c$ is completely axiomatized by the theory

$x + y = y + x$	A1	$x\tau = x$	T1
$x + (y + z) = (x + y) + z$	A2	$\tau x = \tau x + x$	T2
$x + x = x$	A3	$a(\tau x + y) = a(\tau x + y) + ax$	T3
$x(yz) = (xy)z$	A4		
$(x + y)z = xy + xz$	A5	($a \in A \cup \{\tau\}$)	

The τ-laws T1-T3 originate from MILNER [5], who gave a complete axiomatization for a similar model with prefixing instead of general sequential composition. From these axioms, it is easy to show why the

notion of observation congruence is not preserved under refinement of actions: replacing the action a by the term bc, we obtain $bc(\tau x + y) = bc(\tau x + y) + bcx$ from T3, which obviously is not valid in $G/\simeq^c$. By T3, we *do* find $bc(\tau x + y) = b(c(\tau x + y) + cx)$ which unfortunately denotes a different process.
Apart from the problem with refinement, it was observed in VAN GLABBEEK & WEIJLAND [3] that observation equivalence does not strictly preserve the branching structure of processes. This is because an important feature of a bisimulation (cf. PARK [6]) is missing for τ-bisimulation, which is the property that any computation in the one process corresponds to a computation in the other, in such a way that all intermediate states of these computations correspond as well. However, in observation congruence, when satisfying the second requirement of definition 1.2 one may execute arbitrarily many τ-steps in a graph without worrying about the status of the nodes that are passed in the meantime.
In order to overcome this problem, in [3] a different notion was introduced, which yields a finer equivalence on graphs.

DEFINITION 1.3 Two graphs g and h are *branching equivalent* if there exists a symmetric relation $R \subseteq \text{nodes}(g) \times \text{nodes}(h) \cup \text{node}(h) \times \text{nodes}(g)$ (called a *branching bisimulation*) such that:

1. The roots are related by R
2. If R(r,s) and $r \rightarrow^a r'$ $(a \in A \cup \{\tau\})$, then either $a = \tau$ and R(r',s), or there exists a path $s \Rightarrow s_1 \rightarrow^a s'$ such that $R(r,s_1)$ and $R(r',s')$.

Furthermore, g and h are *branching congruent* if we also have that

3. (root condition) Root nodes are related with root nodes only.

Let us write $R: g \leftrightarrows_b h$ if R is a branching bisimulation between g and h and $R: g \leftrightarrows_{rb} h$ if, in addition, R satisfies the root condition. One can prove that the same equivalence is defined when in definition 1.3 *all* intermediate nodes in $s \Rightarrow s_1$ are required to be related with r. Furthermore, observe that a branching bisimulation can also be defined as in definition 1.2, with as extra requirements that $R(r,s_1)$ and $R(r',s_2)$.
It can be proved that branching equivalence and branching congruence are equivalence relations on G. Furthermore, the latter is the coarsest congruence contained in the former. It was shown in [3] that with respect to closed terms, the model $G/\leftrightarrows_{rb}$ is completely axiomatized by the axioms A1-A5 together with

$$x\tau = x \qquad \text{B1}$$
$$x(\tau(y + z) + y) = x(y + z) \qquad \text{B2.}$$

Note that the axioms B1-B2 when applied from left to right only eliminate occurrences of τ's. Using this property, it can be shown that the associated term rewriting system on $G/\equiv_{A1\text{-}A5}$, i.e. G modulo equality induced by the axioms A1-A5, is confluent and terminating. So any two closed branching congruent terms can be reduced to the same normal form.

2. REFINEMENT

In this section we will prove that branching congruence *is* preserved under refinement of actions, and so it allows us to look at actions as abstractions of much larger structures. Consider the following definitions.

DEFINITION 2.1 (substitution)

Let $r: A \to G$ be a mapping from observable actions to graphs, and suppose $g \in G$. Then, the graph $r(g)$ can be found as follows.

For every edge $r \to^a r'$ $(a \in A)$ in g, take a copy $\underline{r(a)}$ of $r(a)$ $(\in G)$. Next, identify r with the root node of $\underline{r(a)}$, and r' with all endnodes of $\underline{r(a)}$, and remove the edge $r \to^a r'$.

Note that in this definition it is never needed to identify r and r', since graphs from G are non-trivial. This way, the mapping r is defined on the domain G. Note that since $\tau \notin A$, τ-edges cannot be substituted by graphs. Finally, observe that every node in g is a node in $r(g)$.

DEFINITION 2.2 (preservation under refinement of actions)

An equivalence $\approx$ on G is said to be *preserved under refinement of actions* if for every mapping $r: A \to G$, we have: $g \approx h \Rightarrow r(g) \approx r(h)$.

In other words, an equivalence $\approx$ is preserved under refinement if it is a congruence with respect to every substitution operator r.

Starting from a relation R: $g \leftrightarrows_{rb} h$, we construct a branching bisimulation relation r(R): $r(g) \leftrightarrows_{rb} r(h)$, proving that preserving branching congruence, every edge with a label from A can be replaced by a graph.

DEFINITION 2.3 Let $r: A \to G$ be a mapping from observable actions to graphs, $g,h \in G$ and R: $g \leftrightarrows_{rb} h$.

Now r(R) is the smallest relation between nodes of $r(g)$ and $r(h)$, such that:

1. $R \subseteq r(R)$.
2. If $r \to^a r'$ and $s \to^a s'$ $(a \in A)$ are edges in g and h such that R(r,s) and R(r',s'), and both edges are replaced by copies $\underline{r(a)}$ and $\overline{r(a)}$ of $r(a)$ respectively, then nodes from $\underline{r(a)}$ and $\overline{r(a)}$ are related by r(R), only if they are copies of the same node in $r(a)$.

Edges $r \to^a r'$ and $s \to^a s'$ $(a \in A)$ such that R(r,s) and R(r',s'), will be called *related* by R, as well as the copies $\underline{r(a)}$ and $\overline{r(a)}$ that are substituted for them. Observe, that on nodes from g and h the relation r(R) is equal to R. Note that if r(R)(r,s), then r is a node in g iff s is a node in h.

THEOREM (refinement)

Branching congruence is preserved under refinement of actions.

PROOF We prove that R: $g \leftrightarrows_{rb} h \Rightarrow$ r(R): $r(g) \leftrightarrows_{rb} r(h)$ by checking the requirements.

1. The root nodes of $r(g)$ and $r(h)$ are related by r(R).
2. Assume r(R)(r,s) and in $r(g)$ there is an edge $r \to^a r'$. Then there are two possibilities (similarly in case $r \to^a r'$ stems from $r(h)$):

(i) The nodes r and s originate from g and h. Then R(r,s), and by the construction of $r(g)$ we find that either $a=\tau$ and $r \to^\tau r'$ was already an edge in g, or g has an edge $r \to^b r^*$ and $r \to^a r'$ is a copy of an initial edge from r(b). In the first case it follows from R: $g \leftrightarrows_{rb} h$ that either R(r',s) - hence r(R)(r',s) - or in h there is a path $s \Rightarrow s_1 \to^\tau s'$ such that $R(r,s_1)$ and R(r',s'). By definition, the same

path also exists in r(h), and we have r(R)(r,s_1) and r(R)(r',s'). In the second case there must be a path $s \Rightarrow s_1 \rightarrow^b s^*$ in h such that R(r,s_1) and R(r*,s*). Then, in r(h) we find a path $s \Rightarrow s_1 \rightarrow^a s'$ (by replacing $\rightarrow^b$ by r(b)) such that r(R)(r,s_1) and r(R)(r',s').

(ii) The nodes r and s originate from related copies $\underline{r(b)}$ and $\overline{r(b)}$ of a substituted graph r(b) (for some $b \in A$), and are no copies of root or endnodes in r(b). Then $r \rightarrow^a r'$ is an edge in $\underline{r(b)}$. From r(R)(r,s) we find that r and s are copies of the same node from r(b). So, there is an edge $s \rightarrow^a s'$ in $\overline{r(b)}$ where s' is a copy of the node in r(b), corresponding with r'. Clearly r(R)(r',s').

3. Since for nodes from g and h we have r(R)(r,s) iff R(r,s), the root condition is satisfied. □

With respect to closed terms, the refinement theorem can be proved much easier by syntactic analysis of proofs, instead of working with equivalences between graphs. For observe that the axioms A1-A5 + B1-B2, that form a complete axiomatization of branching congruence for closed terms, do *not* contain any occurrences of (atomic) actions from A. Now assume we have a proof of some equality s=t between closed terms, then this proof consists of a sequence of applications of axioms from A1-A5 + B1-B2. Since all these axioms are universal equations without actions from A, the actions from s and t can be replaced by general variables, and the proof will still hold. Hence, every equation is an instance of a universal equation *without* any actions. Immediately we find that we can substitute arbitrary closed terms for these variables, obtaining refinement for closed terms.

Nevertheless, the semantic proof of the refinement theorem is important as one may wish to generalize the result to models of larger graphs than just finite ones from *G*.

REFERENCES

[1] J.A.BERGSTRA & J.W.KLOP, *Algebra of communicating processes with abstraction*, TCS 37 (1), pp.77-121, 1985.

[2] L.CASTELLANO, G.DE MICHELIS & L.POMELLO, *Concurrency vs Interleaving: an instructive example*, Bulletin of the EATCS 31, pp.12-15, 1987.

[3] R.J.VAN GLABBEEK & W.P.WEIJLAND, *Branching time and abstraction in bisimulation semantics* (extended abstract), Report CS-R8911, Centrum voor Wiskunde en Informatica, Amsterdam 1989, to appear in: proc. IFIP 11th World Computer Congress, San Francisco 1989.

[4] L.LAMPORT, *On interprocess communication. Part 1: Basic formalism*, Distributed Computing 1 (2), pp.77-85, 1986.

[5] R.MILNER, *A calculus of communicating systems*, Springer LNCS 92, 1980.

[6] D.PARK, *Concurrency and automata on infinite sequences*, proc. 5th GI conf. on Th. Comp. Sci. (P.Deussen ed.), Springer LNCS 104, pp.167-183, 1981.

[7] V.R.PRATT, *Modelling concurrency with partial orders*, International Journal of Parallel Programming 15 (1), pp.33-71, 1986.

Dialectical Program Semantics

παλιντονος αρμονιη · παλιντροπος αρμονιη

Robert E. Kent

Introduction

Dynamic logic [Kozen] seeks to bring dynamic notions into logic and program semantics by basing this semantics and logic on the notion of "predicate transformer". The alternate program semantics of Hoare-style "precondition/postcondition assertions" is usually viewed as a special case of dynamic logic. Dialectical logic [Kent] seeks to bring dynamic notions into logic by basing logic [Lawvere] on the notion of "dialectical contradiction" or *adjoint pair*. How do these three logics connect together? This paper will show that dynamic logic and Hoare-style precondition/postcondition assertional semantics are exactly equivalent, and that dialectical logic subsumes both in the sense that "dynamic logic is the standard aspect of dialectical logic". More particularly, I show in this paper that the axioms of dynamic logic (or alternatively, precondition/postcondition assertional axioms) characterize precisely the dialectical logic notion of *dialectical flow category* (or alternatively, *assertional category*, a notion related but not equivalent to Manes's assertional category [Manes]). A dialectical flow category is a kind of *indexed adjointness* or *dialectical base* which itself is a dialectical enrichment of the notion of *indexed preorder* [Hyland]. In fact, a dialectical flow category is an indexed adjointness of *subtypes* which is locally cartesian closed. The indexing category here is the enriched notion of a *join bisemilattice* [Kent]. Dialectical flow categories objectivize the intuitive idea of predicate transformation or the "dialectical flow of predicates".

Assertional and Flow Categories

In this section we discuss the semantic structures appropriate for dialectical program semantics. The natural axiomatization indicated by these semantic structures, which is an alternate axiomatization of dynamic logic and expressed principally in terms of adjunctions, will be given in the full paper.

A *biposet* is another name for an ordered category; that is, a category $\mathbf{P} = \langle \mathbf{P}, \preceq, \otimes, \mathrm{Id} \rangle$ whose homsets are posets under *term entailment* $\preceq$ and whose composition $\otimes$ called *tensor product* is monotonic on left and right. We prefer to view biposets as vertical structures, preorders with a tensor product, rather than as horizontal structures, ordered categories. The structural aspect of the semantics of dialectical logic is defined in terms of bisemilattices. A *join bisemilattice* or *semiexact biposet* $\mathbf{P} = \langle \langle \mathbf{P}, \preceq, \otimes, \mathrm{Id} \rangle, \oplus, 0 \rangle$ is a biposet whose homsets are finitely complete join-semilattices with *join* terms $s \oplus r$ and *bottom* term $0_{y,x}$ and whose composition (tensor product) is finitely join-continuous. $\mathbf{P}$-objects are called *types* and $\mathbf{P}$-arrows are called *terms*. Any distributive lattice is a one-object join bisemilattice, where tensor product coincides with lattice meet $s \otimes r \stackrel{\mathrm{df}}{=} s \wedge r$. A *morphism of join bisemilattices* $\mathbf{P} \stackrel{H}{\to} \mathbf{Q}$ is a functor which preserves homset order and finite homset joins. A *complete Heyting category*, abbreviated *cHc*, is the same as a complete join

bisemilattice; that is, a join bisemilattice $\mathbf{H}$ whose homsets are complete join semilattices (arbitrary joins exist) and whose tensor product is join continuous (completely distributive w.r.t. joins). Since the homset $\mathbf{H}[x,z]$ is a complete lattice, and left tensor product $r\otimes$ is continuous, it has (and determines) a right adjoint $r \otimes s \preceq_{y,z} t$ iff $s \preceq_{x,z} r \backslash t$ called *left tensor implication*. Similarly, the right tensor product $\otimes r$ has (and determines) a right adjoint $t \otimes r \preceq_{x,z} s$ iff $t \preceq_{x,y} s/r$ called *right tensor implication*. The category **Rel** (also denoted **Mfn**) of sets and binary relations (multivalued functions) is a cHc. Given an alphabet A, the category of formal A-languages $\mathcal{P}(A^*)$ is a one-object cHc (complete Heyting monoid), whose terms are formal languages, whose tensor product is language concatenation, and whose identity is singleton empty string $\{\varepsilon\}$. More generally, every biposet $\mathbf{P}$ has an associated *closure subset category* $\mathcal{P}(\mathbf{P})$ which is a cHc: objects are $\mathbf{P}$-types, arrows are subsets of $\mathbf{P}$-terms $y \xrightarrow{R} x$ when $R \subseteq \mathbf{P}[y,x]$, and homset order is the closed-below order $S \preceq R$ when $S \subseteq \downarrow(R)$. Since every category $\mathbf{C}$ is a biposet with the identity order on homsets, the subset construction $\mathcal{P}(\mathbf{C})$ is a special case of the closure subset construction.

For any type x in a join bisemilattice $\mathbf{P}$ a *comonoid* u at x, denoted by $u{:}x$, is an endoterm $x \xrightarrow{u} x$ which satisfies "coreflexivity" $u \preceq_{x,x} x$, stating that u is a "subpart" of the type (identity term) x, and "cotransitivity" $u \preceq_{x,x} u \otimes u$. Since $u \otimes u \preceq x \otimes u = u$, we can replace the cotransitivity condition with the equality $u \otimes u = u$, which states that u is an "idempotent" term at type x. Comonoids are generalized subtypes. Comonoids of type x are ordered by entailment $\preceq_x \stackrel{\text{df}}{=} \preceq_{x,x}$. The bottom endoterm $0_x \stackrel{\text{df}}{=} 0_{x,x}$ is the smallest comonoid of type x. The join $v \oplus u$ of any two comonoids $v{:}x$ and $u{:}x$ of type x is also a comonoid of type x. Denote the join semilattice of comonoids of type x by $\Omega(x)$. **[Standardization property:]** $\Omega(x)$ is closed under tensor product; in fact, $\Omega(x)$ is a lattice, with the tensor product $v \otimes u$ of two comonoids $v,u \in \Omega(x)$ being the lattice meet in $\Omega(x)$, and the tensor product identity (or type) endoterm x being the largest comonoid of type x. Furthermore, the meet distributes over the join. This standardization property means that the local contexts (monoidal semilattices) of comonoids $\{\Omega(x) \mid x \text{ a type}\}$ are standard contexts (distributive lattices), and shows why propositions (interpreted as comonoids) and programs (interpreted as terms) are subsumed by a single concept. In subset categories $\mathcal{P}(\mathbf{C})$ a comonoid of type x is either the empty endoterm $x \xrightarrow{\emptyset} x$ or the identity singleton $x \xrightarrow{\{x\}} x$, and these can be interpreted as the truth-values **false** and **true**, so that $\Omega(x)$ is the complete Heyting algebra $\Omega(x) \cong \mathbf{2}$.

A *Hoare triple* or *Hoare assertion* $v{:}y \xrightarrow{r} u{:}x$ in a join bisemilattice $\mathbf{P}$, denoted traditionally although imprecisely by $\{v\}r\{u\}$, consists of a "flow specifying" $\mathbf{P}$-term $y \xrightarrow{r} x$ and two $\mathbf{P}$-comonoids, a "precondition" or source comonoid $v \in \Omega(y)$ and a "postcondition" or target comonoid $u \in \Omega(x)$, which satisfy the "precondition/postcondition constraint" $v \otimes r \preceq r \otimes u$. Composition of Hoare triples $\{w\}s\{v\} \otimes \{v\}r\{u\} = \{w\}(s \otimes r)\{u\}$ is well-defined and $\{u\}x\{u\}$ is the identity Hoare triple at the comonoid $u{:}x$. Also, there is a zero triple $\{v\}0_{y,x}\{u\}$ for any precondition $v \in \Omega(y)$ and postcondition $u \in \Omega(x)$, and if $\{v\}r\{u\}$ and $\{v\}s\{u\}$ are two triples with the same precondition and postcondition then $\{v\}(r \oplus s)\{u\}$ is also a triple. So typed comonoids as objects and Hoare triples as arrows form a join bisemilattice $\mathcal{H}(\mathbf{P})$ called the *Hoare assertional category* over $\mathbf{P}$. There is an obvious underlying type/term functor $\mathcal{H}(\mathbf{P}) \xrightarrow{T_P} \mathbf{P}$ which is a morphism of join bisemilattices. For each type x in $\mathbf{P}$, the *fiber* over x is the subcategory $T_P^{-1}(x) \subseteq \mathcal{H}(\mathbf{P})$ of all comonoids and triples which map to x. The objects in $T_P^{-1}(x)$ are the comonoids of type x and the triples in $T_P^{-1}(x)$ are of the form $\{u'\}x\{u\}$, pairs of comonoids of type x satisfying $u' \preceq u$. Hence, the fiber over x is just the join semilattice (actually, distributive lattice) of comonoids $T_P^{-1}(x) = \Omega(x)$.

For each type x in $\mathbf{P}$, the lattice of comonoids $\Omega(x)$ is a (one object) join sub-bisemilattice of $\mathbf{P}$, and the inclusion functor $\Omega(x) \xrightarrow{\text{Inc}_x} \mathbf{P}$ is a morphism of join bisemilattices. Tensor product, which is lattice meet in $\Omega(x)$, forms a *local conjunction functor* $\Omega(x) \xrightarrow{\otimes_x} \mathbf{JSL}$ into the category of join semilattices, defined by

$\otimes_x(z) \stackrel{\text{df}}{=} \Omega(x) = \langle \Omega(x), \oplus, 0 \rangle$ and $\otimes_x(u) \stackrel{\text{df}}{=} \Omega(x) \stackrel{(\,)\otimes u}{\longrightarrow} \Omega(x)$. Conjunction is a join semilattice functor. This example is a special case of the following construct. An *indexed join semilattice* $\langle \mathbf{P}, \Box^{(\,)} \rangle$ consists of: 1. a join bisemilattice $\mathbf{P}$, and 2. a join semilattice functor $\mathbf{P} \stackrel{\Box^{(\,)}}{\longrightarrow} \mathbf{JSL}$: (a) $\Box^x$ is a join semilattice for each type x; (b) $\Box^y \stackrel{\Box^r}{\to} \Box^x$ is a morphism of join semilattices for each term $y \stackrel{r}{\to} x$ called the *direct flow* specified by r, with $\Box^r(0) = 0$ and $\Box^r(v \oplus v') = \Box^r(v) \oplus \Box^r(v')$; (c) $\Box^{(\,)}$ is functorial, with $\Box^x = \mathrm{Id}_{\Box^x}$, and $\Box^{s \otimes r} = \Box^s \cdot \Box^r$; (d) $\Box^{(\,)}$ is a join semilattice functor, (i) if $r \preceq s$ then $\Box^r \leq \Box^s$, (ii) $\Box^0 = \bot$, and (iii) $\Box^{r \oplus s} = \Box^r \vee \Box^s$. Equivalently, an indexed join semilattice is a join bisemilattice morphism $\mathbf{H} \stackrel{T}{\to} \mathbf{P}$, which, as a functor, is an indexed category (an opfibration). A *direct flow category* $\langle \mathbf{P}, \Box^{(\,)} \rangle$ is an indexed join semilattice, (3) which is standard on subtypes: (a) $\Box^x$ is a join subsemilattice of comonoids $\Box^x \subseteq \Omega(x) = \langle \Omega(x), \oplus, 0 \rangle$ for each type x; (b) $\Box^{(\,)}$ restricted to x-comonoids is the local conjunction functor $\mathrm{Inc}_x \cdot \Box^{(\,)} = \otimes_x$; that is, subtype direct flow $\Box^x \stackrel{\Box^u}{\to} \Box^x$ is just conjunction $\Box^u(u') = u' \otimes u$ for each comonoid $u \in \Box^x$. Comonoids in $\Box^x$ and conjunction form a direct flow category $\langle \Box^x, \otimes_x \rangle$ for each type x. A *morphism of direct flow categories* $\langle \mathbf{P}, \Box^{P,(\,)} \rangle \stackrel{H}{\longrightarrow} \langle \mathbf{Q}, \Box^{Q,(\,)} \rangle$ is a morphism of join bisemilattices $\mathbf{P} \stackrel{H}{\longrightarrow} \mathbf{Q}$ which preserves flow $H \cdot \Box^{Q,(\,)} = \Box^{P,(\,)}$. So inclusion $\langle \Box^x, \otimes_x \rangle \stackrel{\mathrm{Inc}_x}{\longrightarrow} \langle \mathbf{P}, \Box^{(\,)} \rangle$ is a morphism of direct flow categories.

A join bisemilattice $\mathbf{P}$ has *direct Hoare flow* when for any term $y \stackrel{r}{\to} x$ and any precondition $v \in \Omega(y)$, there is a postcondition $\Box^r(v) \in \Omega(x)$ called the ***strongest postcondition*** of r which satisfies the axiom $\Box^r(v) \preceq u$ iff $v \otimes r \preceq r \otimes u$ iff $(v{:}y \stackrel{r}{\to} u{:}x) \in \mathrm{Ar}(\mathcal{H}(\mathbf{P}))$ or $\Box^r(v) = \bigwedge\{u \in \Omega(x) \mid v \otimes r \preceq r \otimes u\}$ for any postcondition $u \in \Omega(x)$. Also, $\Box^r(\Box^s(w)) \preceq \Box^{s \otimes r}(w)$ for any comonoid $w \in \Omega(z)$. Some identities for the direct flow operator $\Box^{(\,)}$ are: $\Box^u(u') = u' \otimes u$ for all comonoids $u \in \Omega(x)$; $\Box^{s \otimes r}(w) = \Box^r(\Box^s(w))$ for two composable $\mathbf{P}$-terms $z \stackrel{s}{\to} y$ and $y \stackrel{r}{\to} x$. A join bisemilattice $\mathbf{P}$ has *ranges* when for any $\mathbf{P}$-term $y \stackrel{r}{\to} x$ there is a *range postcondition* $\partial_1(r) \in \Omega(x)$ which satisfies the axioms $\partial_1(r) \preceq u$ iff $r \preceq r \otimes u$ and $\partial_1(s \otimes r) = \partial_1(\partial_1(s) \otimes r)$ for any postcondition $u \in \Omega(x)$ and composable $\mathbf{P}$-term $z \stackrel{s}{\to} y$. Some identities for the range operator ∂_1 are: "subtypes are their own range" $\partial_1(u) = u$ for any comonoid $u \in \Omega(x)$; "the range of a subterm is the subterm of the range" $\partial_1(r \otimes u) = \partial_1(r) \otimes u$ for any term $y \stackrel{r}{\to} x$ and any postcondition $u \in \Omega(x)$; and "only zero has empty range" $\partial_1(r) = 0_x$ iff $r = 0_{y,x}$ for any term $y \stackrel{r}{\to} x$. If $\mathbf{P}$ has direct Hoare flow $\Box^{(\,)}$, then it has ranges ∂_1 defined to be the direct flow of the top (identity) precondition $\partial_1(r) \stackrel{\text{df}}{=} \Box^r(y)$ for any term $y \stackrel{r}{\to} x$. Conversely, if $\mathbf{P}$ has ranges, then it has direct Hoare flow defined to be the range of the tensor product (guarded term) $\Box^r(v) \stackrel{\text{df}}{=} \partial_1(v \otimes r)$. A *direct Hoare flow category* is a join bisemilattice which has direct Hoare flow, or equivalently, ranges. A join bisemilattice $\mathbf{P}$ is a direct Hoare flow category iff the associated functor $\mathcal{H}(\mathbf{P}) \stackrel{T_P}{\longrightarrow} \mathbf{P}$ is an indexed join semilattice $\langle \mathcal{H}(\mathbf{P}), T_P, \mathbf{P} \rangle$. In fact, any direct Hoare flow category is a direct flow category.

Summary

The most important improvement made by dialectical logic over dynamic logic is in the correct and rigorous treatment of subtypes. It is a serious conceptual error [Kozen] to view dynamic logic as a two-sorted structure: one sort being programs and the other sort being propositions. The central viewpoint of dialectical logic is that predicates (here called subtypes, or more precisely, *comonoids*) are special local idempotent kinds of programs (here called *terms* or *processes*), which by their idempotent and coreflexive nature form the standard logical structure of Heyting algebra in the intuitionistic case or Boolean algebra in the classical case. The two dynamic logic operations of program sequencing and predicate conjunction are combined into the one (horizontal) dialectical logic operation of *tensor product* of terms, and the two dynamic logic operations of program summing and predicate disjunction are combined into the one (vertical) dialectical

logic operation of *boolean sum*. Now, tensor product and boolean sum are global operations on terms. In addition, dialectical logic has complement operations called *tensor implications* and *tensor negation* [Kent], which are also global. In contrast to these, dialectical program semantics, introduces local complement operations called *boolean implication* and *boolean negation*.

Global products and coproducts of precondition/postcondition assertions are defined in terms of *biproducts* in the indexing category underlying a dialectical flow category. Biproducts model the semantic notion of "type sum". Completely general axioms for *domains-of-definition* and *ranges*, and their negation duals *kernels* and *cokernels*, can be given, which are equivalent to predicate transformer axioms, and do *not* require the notion of type sum. A nice program semantics has already been given [Manes] which is based upon the notions of *sums* and *bikernels*, but one of the purposes of this paper is to show that dialectical program semantics, the standard logical semantics of "relational structures", does not require sums and only indirectly requires bikernels. *Iterates*, the dialectical logic rendition of the "consideration modality" of linear logic [Girard], are defined as freely generated monoids, and dialectical categories with consideration modality are introduced to ensure the existence of iterates. The important doctrine of linear logic, paraphrased by the statement that "the familiar connective of boolean negation factors into two operations: *linear negation*, which is the purely negative part of negation; and the modality **of_course**, which has the meaning of reaffirmation", is verified in dialectical program semantics, since the local operation of boolean implication (boolean negation) of subtypes factors into the global operation of tensor implication (tensor negation) followed by comonoidal *support*, the dialectical logic rendition of the "affirmation modality" of linear logic. Term hom-set completeness defines the notion of *topology of subtypes*, thereby making further contact with the affirmation modality. In such complete semantics, topologized matrices of terms are defined and shown to be (categorically) equivalent to single terms via the inverse operations of "partitioning" and "summing". With the introduction of type sums a nontopological matrix theory is developed, where ordinary matrices of terms are defined and shown to be (categorically) equivalent to terms with biproducts.

In summary, with dialectical program semantics we hope to unify small-scale and large-scale program semantics by giving a concrete foundation for the observation that "precondition/postcondition assertions are similar in structure to relational database constraints". I am now exploring the close connection between the functional aspect of dialectical program semantics and Martin-Löf type theory given via locally cartesian closed categories [Seeley]. Furthermore, there is a strong connection between dialectical program semantics and algebraic and temporal logic models of regulation in feedback control systems [Wonham].

References

[Girard] J.Y. Girard, Linear Logic. Theoretical Computer Science 50 (1987).

[Hoare] C.A.R. Hoare, An Axiomatic Basis for Computer Programming. Comm. ACM 12 (1967).

[Hyland] J.M.E. Hyland, et al, Tripos Theory. Math. Proc. Camb. Phil. Soc. 88 (1980).

[Kent] R.E. Kent, The Logic of Dialectical Processes. 4th Workshop on MFPS, Univ. Col. (1988).

[Kozen] D. Kozen and J. Tiuryn, Logics of Programs. TR CS-87-172 CS Dept., Wash. St. Univ. (1987).

[Lawvere] F.W. Lawvere, Adjointness in Foundations. Dialectica 23 (1969).

[Manes] E. Manes, Assertional Categories. 3rd Workshop on MFPS, Tulane (1987).

[Pratt] V.R. Pratt, Semantical Considerations on Floyd-Hoare Logic. 17th IEEE Symp. Found. CS. (1976).

[Seeley] R.A.G. Seeley, Locally Cartesian Closed Categories and Type Theory. Camb. Phil. Soc. 95 (1984).

[Wonham] W.M. Wonham, Logic and Language in Control Theory. 25th Allerton Conf. (1987).

Author Index

Baeten, J. 35
Department for Software Technology
Centre for Mathematics and
Computer Science
P.O.Box 4079, 1009 AB Amsterdam
The Netherlands

Benson, D.B. 47
Computer Science Dept.
Washington State University
Pullman, WA 99164-1210

Bidoit, M. 77
Laboratoire de Recherche en Informatique
C.N.R.S. U.A. 410 "Al Khowarizmi"
Universite Paris-Sud - Bat. 490
F - 91405 ORSAY Cedex
France

Bloom, B. 133
NE43-326
MIT Lab. for Computer Science
545 Technology Square
Cambridge, Massachusetts 02139

Bradley, L. 89
Computer Science and Engineering Department
University of California, San Diego
Mail Code C-104
La Jolla, CA 92093

Constable, R. L. 123
Department of Computer Science
306 Upson Hall
Cornell University
Ithaca, NY 14850

Crew, R.F. 39
13Z Manzanita Park
Stanford, CA 94305

Dauchet, M. 181
LIFL (URA 369-CNRS)
Universite de Lille-Flandres-Artois
UFR IEEA
59655 VILLENEUVE D'ASCQ Cedex
France

Ehrig, H. 85
TU Berlin
Institut für Software und
Theoretische Informatik
Franklinstrasse 28/29
D-1000 Berlin 10

Even, S. 185
Computing and Info. Science Dept.
Kansas State University
Manhattan, KS 66506

Everett, R.P. 119
Research & Technology
British Telecom Research Laboratories
Martlesham Heath
IPSWICH, IP5 7RE, England

Fey, W. 85
TU Berlin
Institut für Software und
Theoretische Informatik
Franklinstrasse 28/29
D-1000 Berlin 10

Hansen, H. 85
TU Berlin
Institut für Software und

Theoretische Informatik
Franklinstrasse 28/29
D-1000 Berlin 10

Hatcher, W.S. 125
1060 Brown Avenue
Quebec City, Quebec
Canada G1S 2Z9

Ionescu, D. 115
University of Ottawa
Department of Electrical Engineering
770 King Edward Ave.
Ottawa, Ontario, Canada, K1N 6N5

Iyer, R.R. 47
Computer Science Dept.
Washington State University
Pullman, WA 99164-1210

Jacobs, D. 85
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0782

Janicki, R. 141
Department of Computer Science and Systems
McMaster University
1280 Main Street West
Hamilton, Ont., Canada L8S 4K1

Kent, R.E. 203
Department of Computer Science
University of Arkansas
Little Rock, AR 72207

Kuhl, J. 111
University of Iowa
Department of Electrical Engineering
Iowa City, IA 52242

Lawvere, W.F. 51
State University of New York at Buffalo
Department of Mathematics
Natural Science and Mathematics
106 Diefendorf Hall
Buffalo, New York 14214-3093

Logrippo, L. 107
University of Ottawa
Computer Science Department
Protocols Research Group
Ottawa, Ont. Canada K1N 9B4

Löwe, M. 85
TU Berlin
Institut für Software und
Theoretische Informatik
Franklinstrasse 28/29
D-1000 Berlin 10

Manca, V. 137
University of Pisa
Dip. Informatica
Corso Italia 40
I-56100 Pisa, Italy

Martin, G.A.R. 119
Research & Technology
British Telecom Research Laboratories
Martlesham Heath
IPSWICH, IP5 7RE, England

Meseguer, J. 105
Computer Science Laboratory
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025

Miller, S. 111
University of Iowa
Department of Computer Science
McLean Hall

Iowa City, IA 52242

Müldner, T. 141
University of Western Ontario
Department of Computer Science
48 Edgar Drive, London
Ontario, Canada N6A 1K1

Nivat, M. 9
Tour 45-55 - 5eme Etage
2, Place Jussieu
75251 Paris Cedex 05
France

Oguztuzun, H.M. 195
University of Iowa
Department of Computer Science
McLean Hall
Iowa City, IA 52242

Parpucea, I. 101
University of Cluj-Napoca
Str. M. Kogalniceanu 1
3400 Cluj-Napoca
Romania

Pigozzi, D. 43
Department of Mathematics
Iowa State University
400 Carver Hall
Ames, Iowa 50011

Pratt, V. 177
Computer Science Department
Stanford University
2215 Old Page Mill Road
Palo Alto, CA 94304

Rattray, C.I.M. 165
Department of Computing Science
University of Stirling
Stirling, Scotland FK9 4LA
Great Britain

Riecke, J.G. 133
NE43-326
MIT Lab. for Computer Science
545 Technology Square
Cambridge, Massachusetts 02139

Salibra, A. 137
University of Pisa
Dip. Informatica
Corso Italia 40
I-56100 Pisa, Italy

Schmidt, D. 185
Computing and Info. Science Dept.
Kansas State University
Manhattan, KS 66506

Scollo, G. 137
Department Informatica
University of Twente
P.O.Box 217
NL-7500AE Enschede
The Netherlands

Talcott, C.L. 95
Stanford University
Department of Computer Science
Stanford, CA 94305-2095

Tison, S. 181
LIFL (URA 369-CNRS)
Universite de Lille-Flandres-Artois
UFR IEEA
59655 VILLENEUVE D'ASCQ Cedex
France

Tonga, M. 125
Université d'Yaoundé
Cameroun

Van Glabbek, R.J. 197
Centre for Mathematics and
Computer Science
P.O.Box 4079, 1009 AB Amsterdam
The Netherlands

Vidal, D. 189
C.R.I.N.
BP: 239, 54506 Nancy Cedex
France

Wagner, E. G. 145
IBM T. J. Watson Research Center
P.O. Box 218
Yorktown Hights, NY 10598

Wells, C. 173
Department of Mathematics and Statistics
Case Western Reserve University
Cleveland, Ohio 44106

Wen, L. 115
University of Ottawa
Department of Electrical Engineering
770 King Edward Ave.
Ottawa, Ontario, Canada, K1N 6N5

Weijland, W.P. 197
Centre for Mathematics and
Computer Science
P.O.Box 4079, 1009 AB Amsterdam
The Netherlands

Zhang, H. 129
Department of Computer Science
The University of Iowa
Iowa City, IA 52242